Constraint Satisfaction Problems (Backtracking Search)

• Chapter 6
  – 6.1: Formalism
  – 6.2: Constraint Propagation
  – 6.3: Backtracking Search for CSP
  – 6.4 is about local search which is a very useful idea but we won’t cover it in class.
Acknowledgements

• Much of the material in the lecture slides comes from Fahiem Bacchus, Sheila McIlraith, and Craig Boutilier.

• Some slides come from a tutorial by Andrew Moore via Sonya Allin.

• Some slides are modified or unmodified slides provided by Russell and Norvig.
Constraint Satisfaction Problems (CSP)

• The search algorithms we discussed so far had no knowledge of the states representation (black box).
  – For each problem we had to design a new state representation (and embed in it the sub-routines we pass to the search algorithms).

• Instead we can have a **general state representation** that works well for many different problems.

• We can then build specialized search algorithms that operate efficiently on this general state representation.

• We call the class of problems that can be represented with this specialized representation: CSPs – Constraint Satisfaction Problems.
• The idea: represent states as a vector of feature values.
  – $k$-features (or variables)
  – Each feature takes a value. Each variable has a domain of possible values:
    • height = \{short, average, tall\},
    • weight = \{light, average, heavy\}

• In CSPs, the problem is to search for a set of values for the features (variables) so that the values satisfy some conditions (constraints).
  – i.e., a goal state specified as conditions on the vector of feature values.
Example: Sudoku

2  6  3
7  4  8
6  4  1

8  5  7
1  9  3
4  2  8

1  2  6
8  9  5
3  7  4

4  5  7
9  8  3
6  1  2

2  6  9
3  1  4
5  4  8

7  3  1
Example: Sudoku

• 81 variables, each representing the value of a cell.

• Values: a fixed value for those cells that are already filled in, the values {1-9} for those cells that are empty.

• Solution: a value for each cell satisfying the constraints:
  – No cell in the same column can have the same value.
  – No cell in the same row can have the same value.
  – No cell in the same sub-square can have the same value.
• More formally, a CSP consists of
  – A set of variables $V_1, \ldots, V_n$
  – For each variable a domain of possible values $\text{Dom}[V_i]$.
  – A set of constraints $C_1, \ldots, C_m$.

  – A solution to a CSP is an assignment of a value to all of the variables such that every constraint is satisfied.
  – A CSP is not satisfiable, if no solution exists.
Formalization of a CSP

• Each variable can be assigned any value from its domain.
  • \( V_i = d \) where \( d \in \text{Dom}[V_i] \)

• Each constraint \( C \)
  – Has a set of variables it is over, called its \textit{scope};
    • e.g., \( C(V1,V2,V4) \) ranges over \( V1, V2, V4 \)
  – Has a restriction on the values of the variables in the scope;
    • e.g. \( C(V1,V2,V3) = \langle (V1,V2,V3), V1 \neq V2 \land V1 \neq V4 \land V2 \neq V4 \rangle \)
      or (shorter) \( C(V1,V2,V3): V1 \neq V2, V1 \neq V4, V2 \neq V4 \)
  – Is a Boolean function that maps assignments to the variables in its scope to true/false.
    • e.g. \( C(V1=a,V2=b,V4=c) = \text{True} \)
      – this set of assignments satisfies the constraint.
    • e.g. \( C(V1=b,V2=c,V4=c) = \text{False} \)
      – this set of assignments falsifies the constraint.
Formalization of a CSP

- **Unary** Constraints (over one variable)
  - e.g. \( C(X): X=2; \ C(Y): Y>5 \)
- **Binary** Constraints (over two variables)
  - e.g. \( C(X,Y): X+Y<6 \)
  - Can be represented by **Constraint Graph**
    - Nodes are variables, arcs show constraints.
    - e.g. 4-Queens:

- **Higher-order** constraints: over 3 or more variables
  - We can convert any constraint into a set of binary constraints (may need some auxiliary variables).
  - Look at the exercise in the book.
Example: Sudoku

- **Variables:** \( V_{11}, V_{12}, \ldots, V_{21}, V_{22}, \ldots, V_{91}, \ldots, V_{99} \)

- **Domains:**
  - \( \text{Dom}[V_{ij}] = \{1-9\} \) for empty cells
  - \( \text{Dom}[V_{ij}] = \{k\} \) a fixed value \( k \) for filled cells.

- **Constraints:**
  - **Row Constraints:**
    - \( CR1(V_{11}, V_{12}, V_{13}, \ldots, V_{19}) \)
    - \( CR2(V_{21}, V_{22}, V_{23}, \ldots, V_{29}) \)
    - \( \ldots, CR9(V_{91}, V_{92}, \ldots, V_{99}) \)
  - **Column Constraints:**
    - \( CC1(V_{11}, V_{21}, V_{31}, \ldots, V_{91}) \)
    - \( CC2(V_{21}, V_{22}, V_{13}, \ldots, V_{92}) \)
    - \( \ldots, CC9(V_{19}, V_{29}, \ldots, V_{99}) \)
  - **Sub-Square Constraints:**
    - \( CSS1(V_{11}, V_{12}, V_{13}, V_{21}, V_{22}, V_{23}, V_{31}, V_{32}, V_{33}) \)
    - \( CSS1(V_{14}, V_{15}, V_{16}, \ldots, V_{34}, V_{35}, V_{36}) \)
• Each of these constraints is over 9 variables, and they are all the same constraint:
  – Any assignment to these 9 variables such that each variable has a unique value satisfies the constraint.
  – Any assignment where two or more variables have the same value falsifies the constraint.

• Special kind of constraints called ALL-DIFF constraints.
  – An ALL-DIFF constraint over k variables can be equivalently represented by \((k \text{ choose } 2)\) “not-equal constraints” (NEQ) over each pair of these variables.
  – e.g. \(\text{CSS1}(V_{11}, V_{12}, V_{13}, V_{21}, V_{22}, V_{23}, V_{31}, V_{32}, V_{33}) = \text{NEQ}(V_{11}, V_{12}), \text{NEQ}(V_{11}, V_{13}), \text{NEQ}(V_{11}, V_{21}), ..., \text{NEQ}(V_{32}, V_{33})\)
  – Remember: all higher-order constraints can be converted into a set of binary constraints

Example: Sudoku
• Thus Sudoku has 3x9 ALL-DIFF constraints, one over each set of variables in the same row, one over each set of variables in the same column, and one over each set of variables in the same sub-square.
Solving CSPs

• CSPs can be solved by a specialized version of depth-first search.
  – Actually depth-limited search. Why?

• Key intuitions:
  – We can build up to a solution by searching through the space of partial assignments.
  – Order in which we assign the variables does not matter – eventually they all have to be assigned. We can decide on a suitable value for one variable at a time!
  ➔ This is the key idea of backtracking search.
  – If during the process of building up a solution we falsify a constraint, we can immediately reject all possible ways of extending the current partial assignment.
CSP as a Search Problem

- **Initial state**: empty assignment
- **Successor function**: a value is assigned to any unassigned variable, which does not conflict with the currently assigned variables
- **Goal test**: the assignment is complete
- **Path cost**: irrelevant
Solving CSPs – Backtracking Search

• **Bad news**: 3SAT is a finite CSP and known to be NP-complete, so we cannot expect to do better in the worst case

• Backtracking Search: DFS with single-variable assignments for a CSP
  – Basic uninformed search for solving CSPs
  – Gets rid of unnecessary permutations in search tree and significantly reduces search space:
    • Time complexity: reduction from $O(d^n!)$ to $O(d^n)$
      $d$ ... max. number of values of some variable (branching factor)
      $n$ ... number of variables (depth)
    • Sudoku example: order of filling a square does not matter
      – [...] $(2,3)=7, (3,3)=8, ...$ = [...] $(3,3)=8, (2,3)=7, ...$
      – $9^{81}$ states instead of $9^{81!}$ states
Backtracking Search: The Algorithm BT

• These ideas lead to the backtracking search algorithm

BT(Level)

If all variables assigned
    PRINT Value of each Variable
    RETURN or EXIT (RETURN for more solutions)
    (EXIT for only one solution)

V := PickUnassignedVariable()
Variable[Level] := V
Assigned[V] := TRUE

for d := each member of Domain(V) (the domain values of V)
    Value[V] := d
    for each constraint C such that V is a variable of C
        and all other variables of C are assigned:
            IF C is not satisfied by the set of current
            assignments: BREAK;
            ELSE BT(Level+1)

return
Backtracking Search

- The algorithm searches a tree of partial assignments.

Children of a node are all possible values of some (any) unassigned variable.

The root has the empty set of assignments.

Search stops descending if the assignments on path to the node violate a constraint.
Backtracking Search

• Heuristics are used to determine
  – the order in which variables are assigned:
    PickUnassignedVariable()
  – the order of values tried for each variable.

• The choice of the next variable can vary from branch to branch, e.g.,
  – under the assignment V1=a we might choose to assign V4 next, while under V1=b we might choose to assign V5 next.

• This “dynamically” chosen variable ordering has a tremendous impact on performance.
Example: N-Queens

• Place N Queens on an N x N chess board so that no Queen can attack any other Queen.
Example: N-Queens

• Problem formulation:
  – N variables (N queens)
  – $N^2$ values for each variable representing the positions on the chessboard
Example: N-Queens

- $Q_1 = 1$, $Q_2 = 15$, $Q_3 = 21$, $Q_4 = 32$, $Q_5 = 34$, $Q_6 = 44$, $Q_7 = 54$, $Q_8 = 59$
Example: N-Queens

• This representation has \((N^2)^N\) states (different possible assignments in the search space)
  – For 8-Queens: \(64^8 = 281,474,976,710,656\)

• Is there a better way to represent the N-queens problem?
  – We know we cannot place two queens in a single row \(\Rightarrow\) we can exploit this fact in the choice of the CSP representation already
Example: N-Queens

• Better Modeling:
  – N variables Qi, one per row.
  – Value of Qi is the column the Queen in row i is placed; possible values \{1, ..., N\}.

• This representation has \(N^N\) states:
  – For 8-Queens: \(8^8 = 16,777,216\)

• The choice of a representation can decided whether or not we can solve a problem!
Example: N-Queens

- Q1 = 1, Q2 = 7, Q3 = 5, Q4 = 8, Q5 = 2, Q6 = 4, Q7 = 6, Q8 = 3
Example: N-Queens

• Constraints:
  – Can’t put two Queens in same column
    \( Q_i \neq Q_j \) for all \( i \neq j \)
  – Diagonal constraints
    \(|Q_i - Q_j| \neq i-j\)
  • i.e., the difference in the values assigned to \( Q_i \) and \( Q_j \) can’t be equal to the difference between \( i \) and \( j \).
Example: N-Queens
Example: N-Queens
Example: N-Queens
Example: N-Queens
Problems with Plain Backtracking

Sudoku: The 3,3 cell has no possible value.

```
 1 2 3

 4 5 6

 7

 8

 9
```
Problems with Plain Backtracking

• In the backtracking search we won’t detect that the (3,3) cell has no possible value until all variables of the row/column (involving row or column 3) or the sub-square constraint (first sub-square) are assigned. So we have the following situation:

• Leads to the idea of **constraint propagation**
Constraint Propagation

- Constraint propagation refers to the technique of "looking ahead" at the yet unassigned variables in the search.

- Try to detect obvious failures: "Obvious" means things we can test/detect efficiently.

- Even if we don’t detect an obvious failure we might be able to eliminate some possible part of the future search.
Constraint Propagation

• Propagation has to be applied during the search; potentially at every node of the search tree.

• Propagation itself is an inference step which needs some resources (in particular time)
  – If propagation is slow, this can slow the search down to the point where using propagation actually slows search down!
  – There is always a tradeoff between searching fewer nodes in the search, and having a higher nodes/second processing rate.

• We will look at two main types of propagation.
Constraint Propagation: Forward Checking

• Forward checking is an extension of backtracking search that employs a “modest” amount of propagation (look ahead).

• When a variable is instantiated we check all constraints that have only one uninstantiated variable remaining.

• For that uninstantiated variable, we check all of its values, pruning those values that violate the constraint.
Forward Checking Algorithm

• For a single constraint C:

    FCCheck(C, x)
    
    // C is a constraint with all its variables already
    // assigned, except for variable x.
    for d := each member of CurDom[x]
        IF making x = d together with previous assignments
        to variables in scope C falsifies C
        THEN remove d from CurDom[V]
        IF CurDom[V] = {} then return DWO (Domain Wipe Out)
    return ok
Forward Checking Algorithm

**FC**(Level) /*Forward Checking Algorithm*/

If all variables are assigned
- PRINT Value of each Variable
- RETURN or EXIT (RETURN for more solutions) (EXIT for only one solution)

V := PickAnUnassignedVariable()
Variable[Level] := V
Assigned[V] := TRUE
for d := each member of CurDom(V)
  Value[V] := d
  DWOoccurred:= False
  for each constraint C over V that has one unassigned variable in its scope (say X).
    if(FCCheck(C,X) == DWO) /* X domain becomes empty*/
      DWOoccurred:= True /* no point to continue*/
      break
    if(not DWOoccurred) /*all constraints were ok*/
      FC(Level+1)
      RestoreAllValuesPrunedByFCCheck()
return;
4-Queens Problem

- Encoding with Q1, ..., Q4 denoting a queen per column
  - cannot put two queens in same row (instead of same column)
4-Queens Problem

- Forward checking reduced the domains of all variables that are involved in a constraint with one uninstantiated variable:
  - Here all of Q2, Q3, Q4

![Diagram showing the 4-Queens Problem with variables Q1, Q2, Q3, Q4 and their respective domains.](image)
4-Queens Problem

Q1: \{1,2,3,4\}, Q2: \{3,4\}, Q3: \{2,4\}, Q4: \{2,3\}
4-Queens Problem

\[
\begin{array}{cccc}
& 1 & 2 & 3 & 4 \\
1 & & & \star & \\
2 & \bigcirc & & & \\
3 & & \bigcirc & & \\
4 & & & \bigcirc & \\
\end{array}
\]

Q1 \{1,2,3,4\} -> Q2 \{3,4\}
Q3 \{1,2,3,4\} -> Q4 \{2,3,4\}

DWO
4-Queens Problem

- **Q1**: \{1, 2, 3, 4\}
- **Q2**: \{ , , , 4\}
- **Q3**: \{ , 2, , 4\}
- **Q4**: \{ , 2, 3, , \}
4-Queens Problem

Q1 \{1, 2, 3, 4\}
Q2 \{\ , \ , \ , 4\}
Q3 \{\ , 2, \ , \}\nQ4 \{\ , \ , 3, \}
4-Queens Problem

Q1 \{1,2,3,4\}
Q2 \{ , , , 4\}
Q3 \{ , 2, , \}
Q4 \{ , , 3, \}
4-Queens Problem

- Q1: \{1,2,3,4\}
- Q2: \{\text{-},\text{-},\text{-},4\}
- Q3: \{\text{-},2,\text{-},\text{-}\}
- Q4: \{\text{-},\text{-},\text{-},\text{-}\}

DWO
4-Queens Problem

- Exhausted the subtree with $Q1=1$; try now $Q1=2$
4-Queens Problem

Q1 \{2,3,4\}
Q2 \{4\}
Q3 \{1,3\}
Q4 \{1,3,4\}
4-Queens Problem

Q1: \{1, 2, 3, 4\}
Q2: \{4\}
Q3: \{1, 3\}
Q4: \{1, 3\}

Diagram:

```
1 2 3 4
1 . Q Q Q
2 Q . . .
3 Q . . .
4 Q Q . .
```
4-Queens Problem
4-Queens Problem

- **Q1**: \{1, 2, 3, 4\}
- **Q2**: \{1, 2, 3, 4\}
- **Q3**: \{1, 2, 3, \}
- **Q4**: \{1, 2, 3, \}

The diagram shows a 4x4 grid with queens placed at specific positions. Each queen must attack exactly one other queen, and the queens are placed in such a way that they do not attack each other.
4-Queens Problem

- We have now find a solution: an assignment of all variables to values of their domain so that all constraints are satisfied.
FC: Restoring Values

• After we backtrack from the current assignment (in the for loop) we must restore the values that were pruned as a result of that assignment.

• Some bookkeeping needs to be done, as we must remember which values were pruned by which assignment (FCCheck is called at every recursive invocation of FC).
FC: Minimum Remaining Values Heuristics (MRV)

• FC also gives us for free a very powerful heuristic to guide us which variables to try next:
  – Always branch on a variable with the smallest remaining values (smallest CurDom).
  – If a variable has only one value left, that value is forced, so we should propagate its consequences immediately.
  – This heuristic tends to produce skinny trees at the top. This means that more variables can be instantiated with fewer nodes searched, and thus more constraint propagation/DWO failures occur with less work.
  – We can find a inconsistency such as in the Sudoku example much faster.
MRV Heuristic: Human Analogy

• What variables would you try first?

Domain of each variable: 
{1, ..., 9}

(1, 5): impossible values:  
Row: {1, 4, 5, 6, 8}  
Column: {1, 3, 4, 5, 7, 9}  
Subsquare: {5, 7, 9}  
→ Domain = {2}

(9, 5): impossible values:  
Row: {1, 5, 7, 8, 9}  
Column: {1, 3, 4, 5, 7, 9}  
Subsquare: {1, 5, 7, 9}  
→ Domain = {2, 6}

Most restricted variables! = MRV

After assigning value 2 to cell (1,5): Domain = {6}
Example – Map Colouring

• Color the following map using red, green, and blue such that adjacent regions have different colors.
Example – Map Colouring

• Modeling
  – Variables: WA, NT, Q, NSW, V, SA, T
  – Domains: $D_i=\{\text{red, green, blue}\}$
  – Constraints: adjacent regions must have different colors.
    • E.g. WA $\neq$ NT
Example – Map Colouring

- **Forward checking idea**: keep track of remaining legal values for unassigned variables.
- Terminate search when any variable has no legal values.
Example – Map Colouring

• Assign \( \{WA=\text{red}\} \)

• Effects on other variables connected by constraints to \( WA \)
  – \( NT \) can no longer be red
  – \( SA \) can no longer be red
Example – Map Colouring

• Assign \{Q=\text{green}\}
• Effects on other variables connected by constraints with Q
  – NT can no longer be green
  – NSW can no longer be green
  – SA can no longer be green
• MRV heuristic would automatically select NT or SA next
Example – Map Colouring

• Assign \(V=\text{blue}\)

• Effects on other variables connected by constraints with \(V\)
  – NSW can no longer be blue
  – SA is empty

• FC has detected that partial assignment is inconsistent with the constraints and backtracking can occur.
Empirically

- FC often is about 100 times faster than BT
- FC with MRV (minimal remaining values) often 10000 times faster.
- But on some problems the speed up can be much greater
  - Converts problems that are not solvable to problems that are solvable.
- Other more powerful forms of consistency are commonly used in practice.
Constraint Propagation: Arc Consistency

• Another form of propagation: make each arc consistent
  – C(X,Y) is consistent iff for every value of X there is some value of Y that satisfies C.
  – Idea: ensure that every binary constraint is satisfiable (2-consistency)
    • Binary constraints = arcs in the constraint graph
    • Remember: All higher-order constraints can be expressed as a set of binary constraints
Constraint Propagation: Arc Consistency

• Can remove values from the domain of variables:
  – e.g. C(X,Y): X>Y Dom(X)={1,5,11} Dom(Y)={3,8,15}
    • For X=1 there is no value of Y s.t. 1>Y => remove 1 from domain X
    • For Y=15 there is no value of X s.t. X>15, so remove 15 from domain Y
    • We obtain more restricted domains Dom(X)={5,11} and Dom(Y)={3,8}
  – Have to try much fewer values in the search tree.

• Removing a value from a domain may trigger further inconsistency, so we have to repeat the procedure until everything is consistent.
  – For efficient implementation, we keep track of inconsistent arcs by putting them in a Queue (See AC3 algorithm in the book).

• This is stronger than forward checking. why?
• Since NSW loses a value, we need to recheck all constraints involving NSW: other neighbours are Q, V
Arc Consistency – Map Colouring Example

- Since V loses a value, we need to recheck all constraints involving V: other neighbours are SA
- Recheck all constraints involving SA
Arc Consistency – Map Colouring Example

• (SA, NT) is not satisfiable any longer – we detected an unavoidable failure in the assignment {WA=red, Q=green}
  – Forward checking would have detected it as well. Why?
Arc-Consistency – Example

• CSP with 3 variables X, Y, Z
  – Domains:
    • Dom(X) = \{1, \cdots, 10\}
    • Dom(Y) = \{5, \cdots, 15\}
    • Dom(Z) = \{5, \cdots, 20\}

  – Constraints:
    • C(X,Y): X > Y
    • C(Y,Z): Y + Z = 12
    • C(X,Z): X + Z = 16
Arc Consistency – Example

• Draw the constraint graph.

• Are the constraints arc consistent?

C(X,Y) is consistent iff for every value of X there is some value of Y that satisfies C.
Arc Consistency – Example

• Apply arc consistency method repeatedly so they become arc consistent.
  – For $X=1,2,3,4,5$ there is no value of $Y$ s.t. $X > Y$
    => remove $1,2,3,4,5$ from domain $X$
  – For $Y > 7$ there is no value of $Z$ s.t. $Y + Z = 12$
    => remove $8,\ldots,15$ from domain $Y$
  – For $Z > 7$ there is no value of $Y$ s.t. $Y + Z = 12$
    => remove $8,\ldots,20$ from domain $Z$
  – For $X=6,7,8$ there is no value of $Z$ s.t.
    $X + Z = 16$ => remove $6,7,8$ from domain $X$
  – For $Z=5$ there is no value of $X$ s.t.
    $X + Z = 16$ => remove $5$ from domain $Z$
  – For $Y=7$ there is no value of $Z$ s.t.
    $Y + Z = 12$ => remove $7$ from domain $Y$

\[
\begin{align*}
\text{Dom}(X) &= \{9, 10\} \\
\text{Dom}(Y) &= \{5, 6\} \\
\text{Dom}(Z) &= \{6, 7\}
\end{align*}
\]

\[
\begin{align*}
\text{C}(X,Y): X &> Y \\
\text{C}(Y,Z): Y + Z = 12 \\
\text{C}(X,Z): X + Z = 16
\end{align*}
\]
Other forms of Consistency

• **Generalized arc consistency**: for an n-ary constraint, for each value of the domain of a variable, there exists a tuple of values for the remaining n-1 variables that satisfy the constraint.

• **Path consistency**: a pair (V1, V2) is path-consistent with respect to a third variable V3 if for every assignment \(\{V1=a, V2=b\}\) consistent with all binary constraints of \(\{V1, V2\}\), there is an assignment to V3 that satisfies all binary constraints on \(\{V1, V3\}\) and \(\{V2, V3\}\).
  - Think of V3 being on a path of length 2 from V1 to V2.

• **Strong k-consistency**
  - k-consistent, (k-1)-consistent, etc.
  - Very expensive: any algorithm establishing k-consistency requires exponential time and space in k.
  - In practical solvers: 2-consistency, sometimes 3-consistency.
Many real-world applications of CSP

• Assignment problems
  – who teaches what class
• Timetabling problems
  – exam schedule
• Transportation scheduling
• Floor planning
• Factory scheduling
• Hardware configuration
  – a set of compatible components
### CSP Solvers

- Much work on various heuristics for variable and value selection
- Fourth CSP Solver Competition Results 2009, Category: Only binary constraints

<table>
<thead>
<tr>
<th>Rank</th>
<th>Solver</th>
<th>Version</th>
<th>Number of solved instances</th>
<th>Detail</th>
<th>% of all instances</th>
<th>% of VBS</th>
<th>Cumulated CPU time on solved instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mistral</td>
<td>1.545</td>
<td>570</td>
<td>326 SAT, 244 UNSAT</td>
<td>90%</td>
<td>94%</td>
<td>46856.82</td>
</tr>
<tr>
<td>2</td>
<td>Choco2.1.1b</td>
<td>2009-07-16</td>
<td>556</td>
<td>314 SAT, 242 UNSAT</td>
<td>88%</td>
<td>91%</td>
<td>55414.54</td>
</tr>
<tr>
<td>3</td>
<td>Abscon 112v4</td>
<td>AC</td>
<td>551</td>
<td>314 SAT, 237 UNSAT</td>
<td>87%</td>
<td>90%</td>
<td>58656.18</td>
</tr>
<tr>
<td>4</td>
<td>Abscon 112v4</td>
<td>ESAC</td>
<td>547</td>
<td>310 SAT, 237 UNSAT</td>
<td>86%</td>
<td>90%</td>
<td>50388.31</td>
</tr>
<tr>
<td>5</td>
<td>Choco2.1.1</td>
<td>2009-06-10</td>
<td>547</td>
<td>309 SAT, 238 UNSAT</td>
<td>86%</td>
<td>90%</td>
<td>57385.70</td>
</tr>
<tr>
<td>6</td>
<td>Concrete DC</td>
<td>2009-07-14</td>
<td>504</td>
<td>297 SAT, 207 UNSAT</td>
<td>79%</td>
<td>83%</td>
<td>60964.15</td>
</tr>
<tr>
<td>7</td>
<td>Concrete</td>
<td>2009-07-14</td>
<td>503</td>
<td>297 SAT, 206 UNSAT</td>
<td>79%</td>
<td>83%</td>
<td>49380.89</td>
</tr>
<tr>
<td>8</td>
<td>Sugar</td>
<td>v1.14.6+minisat</td>
<td>466</td>
<td>274 SAT, 192 UNSAT</td>
<td>73%</td>
<td>77%</td>
<td>71234.79</td>
</tr>
<tr>
<td>9</td>
<td>Sugar</td>
<td>v1.14.6+picosat</td>
<td>438</td>
<td>270 SAT, 168 UNSAT</td>
<td>69%</td>
<td>72%</td>
<td>53442.74</td>
</tr>
<tr>
<td>10</td>
<td>SAT4J CSP</td>
<td>2.1.1</td>
<td>421</td>
<td>259 SAT, 162 UNSAT</td>
<td>66%</td>
<td>69%</td>
<td>51843.43</td>
</tr>
<tr>
<td>11</td>
<td>bpsolver</td>
<td>09</td>
<td>416</td>
<td>253 SAT, 163 UNSAT</td>
<td>66%</td>
<td>68%</td>
<td>56052.25</td>
</tr>
<tr>
<td>12</td>
<td>pcs-restart</td>
<td>0.3.2</td>
<td>394</td>
<td>241 SAT, 153 UNSAT</td>
<td>62%</td>
<td>65%</td>
<td>46361.44</td>
</tr>
<tr>
<td>13</td>
<td>pcs</td>
<td>0.3.2</td>
<td>393</td>
<td>238 SAT, 155 UNSAT</td>
<td>62%</td>
<td>65%</td>
<td>56915.05</td>
</tr>
</tbody>
</table>

Total number of instances in the category: 635
CSP Solvers

Time to solve an instance
(SAT answers, category 2-ARY-EXT)

CPU time (s)

number of solved instances

Abscon 112v4 AC
Abscon 112v4 ESAC
bpsolver 09
Choco2.1.1 2009-06-10
Choco2.1.1b 2009-07-16
Concrete 2009-07-14
Concrete DC 2009-07-14
Conquer 2009-07-10
Mistral 1.545
pcs 0.3.2
pcs-restart 0.3.2
SAT4J CSP 2.1.1
Sugar v1.14.6+minisat
Sugar v1.14.6+picosat

Time to solve an instance
(UNSAT answers, category 2-ARY-EXT)

CPU time (s)

number of solved instances

Abscon 112v4 AC
Abscon 112v4 ESAC
bpsolver 09
Choco2.1.1 2009-06-10
Choco2.1.1b 2009-07-16
Concrete 2009-07-14
Concrete DC 2009-07-14
Conquer 2009-07-10
Mistral 1.545
pcs 0.3.2
pcs-restart 0.3.2
SAT4J CSP 2.1.1
Sugar v1.14.6+minisat
Sugar v1.14.6+picosat