Date: 3/4/2016

- Major Code Changes
  - Added Heuristic Flags and Arc Consistency
    - Select Next Variable – Added MRV and DH segments

```
Variable SudokuSolver::selectNextVariable (SudokuPuzzle
puzzle) {
    std::vector<Variable> unassigned =
getUnassignedVariables(puzzle);
    if (_flags[HeuristicFlag::kMRV]) {
        std::vector<Variable> mrv = applyMRV
(puzzle,unassigned);
        if (_flags[HeuristicFlag::kDH]) {
            std::vector<Variable> dh = applyDH
(puzzle,mrv);
            return dh[0];
        }
        else {
            return mrv[0];
        }
    }
    else if (_flags[HeuristicFlag::kDH]) {
        std::vector<Variable> dh = applyDH
(puzzle,unassigned);
        return dh[0];
    }
    else if (unassigned.size () > 0) {
        return unassigned[0];
    }
    else {
        Position p(-1,-1);
        Variable noV (p,{});
        return noV;
    }
}
```

    - Order Domain Values – Added LCV segments

```
Domain SudokuSolver::orderDomainValues (SudokuPuzzle
puzzle, Position position) {
    if (_flags[HeuristicFlag::kLCV]) {
        Domain lcv = applyLCV (puzzle,
puzzle.sudoku()[position._x][position._y]);
        return lcv;
    }
    else {
        return
puzzle.sudoku()[position._x][position._y]._domain;
    }
}
```

    - MRV

```
std::vector<Variable>
SudokuSolver::applyMRV(SudokuPuzzle puzzle,
std::vector<Variable> unassigned) {
    int min = getRemainingValues
(puzzle,unassigned[0]._position);
    for (std::size_t i = 1; i < unassigned.size ();
++i) {
```

```
            int cur = getRemainingValues
    (puzzle,unassigned[i]._position);
            if (cur < min) {
                min = cur;
            }
        }
        std::vector<Variable> mrvVector;
        for (std::size_t i = 0; i < unassigned.size ();
    ++i) {
            if (getRemainingValues
    (puzzle,unassigned[i]._position) == min) {
                mrvVector.insert (mrvVector.end (),
    unassigned[i]);
            }
        }
        return mrvVector;
    }
```

- DH

```
std::vector<Variable> SudokuSolver::applyDH(SudokuPuzzle
puzzle, std::vector<Variable> unassigned) {
    int max = getDegree
(puzzle,unassigned[0]._position,unassigned);
    for (std::size_t i = 1; i < unassigned.size (); ++i)
{
        int cur = getDegree
(puzzle,unassigned[i]._position,unassigned);
        if (cur > max) {
            max = cur;
        }
    }
    std::vector<Variable> dhVector;
    for (std::size_t i = 0; i < unassigned.size (); ++i)
{
        if (getDegree
(puzzle,unassigned[i]._position,unassigned) == max) {
            dhVector.insert (dhVector.end (),
unassigned[i]);
        }
    }
    return dhVector;
}
```

- LCV

```
Domain SudokuSolver::applyLCV(SudokuPuzzle puzzle,
Variable variable) {
    std::vector<std::pair<char,int>> lcvVector;
    for (std::size_t i = 0; i <
variable._domain._domain.size(); ++i) {
        std::pair<char,int>
p(variable._domain._domain[i],getConstraints
(puzzle,variable._position,variable._domain._domain[i]))
;
        lcvVector.insert (lcvVector.end(),p);
    }
    for (std::size_t i = 0; i < lcvVector.size(); ++i) {
        int j = i;
```

```
        while (j > 0 && lcvVector[j].second <
lcvVector[j - 1].second) {
            std::pair<char,int> temp = lcvVector[j];
            lcvVector[j] = lcvVector[j - 1];
            lcvVector[j - 1] = temp;
            j--;
        }
    }
    Domain d;
    for (std::size_t i = 0; i < lcvVector.size(); ++i) {
        d.add(lcvVector[i].first);
    }
    return d;
}
```

- AC3

```
bool SudokuSolver::applyAC3 (SudokuPuzzle &puzzle, int
level) {
    std::vector<std::pair<Position,Position>> arcs;
    for (std::size_t x = 0; x < puzzle.n(); ++x) {
        for (std::size_t y = 0; y < puzzle.n(); ++y) {
            std::vector<Variable> neighbors =
getNeighbors (puzzle,puzzle.sudoku()[x][y]._position);
            for (std::size_t j = 0; j <
neighbors.size(); ++j) {
                Position c(x,y);
                std::pair<Position,Position>
p(c,neighbors[j]._position);
                arcs.insert (arcs.end(),p);
            }
        }
    }
    for (std::size_t i = 0; i < arcs.size(); ++i) {
        char fail;
        Position cur1 = arcs[i].first;
        Position cur2 = arcs[i].second;
        arcs.erase (arcs.begin ());
        --i;
        if (!checkArc (puzzle,cur1,cur2,fail)) {
            Domain d;
            d.add(fail);
            int x = cur1._x, y = cur1._y;
            bookKeep(level,puzzle.sudoku()[x][y],d);
            if
(puzzle.sudoku()[x][y]._domain._domain.empty()) {
                return false;
            }
            std::vector<Variable> neighbors =
getNeighbors (puzzle,cur1);
            for (std::size_t j = 0; j <
neighbors.size(); ++j) {
                bool in = false;
                std::pair<Position,Position>
p(neighbors[j]._position,cur1);
                for (std::size_t k = 0; k < arcs.size();
++k) {
```

```
                    if (p.first._x == arcs[k].first._x
&& p.first._y == arcs[k].first._y && p.second._x ==
arcs[k].second._x && p.second._y == arcs[k].second._y) {
                        in = true;
                    }
                }
                if (!in) {
                    arcs.insert (arcs.end (),p);
                }
            }
        }
    }
    return true;
}
```

- Get Degree

```
int SudokuSolver::getDegree (SudokuPuzzle puzzle,
Position position, std::vector<Variable> unassigned) {
    int degree = 0;
    std::vector<Variable> neighbors = getNeighbors
(puzzle, position);
    for (std::size_t i = 0; i < neighbors.size (); ++i)
{
        if (neighbors[i]._value == '0') {
            ++degree;
        }
    }
    return degree;
}
```

- Get Remaining Values

```
int SudokuSolver::getRemainingValues  (SudokuPuzzle
puzzle, Position position) {
    return
puzzle.sudoku()[position._x][position._y]._domain._doma
in.size();
}
```

- Get Constraints

```
int SudokuSolver::getConstraints (SudokuPuzzle puzzle,
Position position, char value) {
    int constraints = 0;
    std::vector<Variable> neighbors = getNeighbors
(puzzle, position);
    for (std::size_t i = 0; i < neighbors.size (); ++i)
{
        if (neighbors[i]._value == '0') {
            for (std::size_t j = 0; j <
neighbors[i]._domain._domain.size(); ++j) {
                if (neighbors[i]._domain._domain[j] ==
value) {
                    ++constraints;
                }
            }
        }
    }
    return constraints;
}
```