CS 4804 Homework 2
Solution Sketches

---

1. **(10 points)** The following constraint graph has width 1.

```
A---B---C---D
```

2. **(10 points)** The following constraint graph has width 2.

```
A---B
|   |
|   |
C---D
```

3. **(10 points)** The following constraint graph has width 3.

```
A---B
|\ /|
| X |
|/ \|
C---D
```

4. **(20 points)** First, lets give the tile positions some names such as:

```
a b c
d e f
g h i
```

Variables: There will be 81 variables, representing whether a given number tile is at a given position, written as follows $x_{\text{position number}}$ (i.e. $x_{a1}, x_{a2}, ..., x_{i9}$). Thus, x_{a1} is true if position a has number 1, and so on.

Clauses: There are a variety of clauses that can be stated for this problem:

$$
\begin{array}{ll}
 & (x_{a1} \vee x_{a2} \vee \cdots \vee x_{a9}) \quad \text{// position a has at least one of the nine numbers} \\
\wedge & (x_{b1} \vee x_{b2} \vee \cdots \vee x_{b9}) \quad \text{// position b has at least one of the nine numbers} \\
\wedge & \cdots \\
\wedge & (x_{i1} \vee x_{i2} \vee \cdots \vee x_{i9}) \quad \text{// position i has at least one of the nine numbers} \\
\wedge & (\overline{x_{a1}} \vee \overline{x_{a2}}) \quad \text{// position a cannot have both 1 and 2} \\
\wedge & (\overline{x_{a1}} \vee \overline{x_{a3}}) \quad \text{// position a cannot have both 1 and 3} \\
\wedge & \cdots \\
\wedge & (\overline{x_{i8}} \vee \overline{x_{i9}}) \quad \text{// position i cannot have both 8 and 9}
\end{array}
$$

The arrangement so far still allows the same number (e.g., 1) to be present in all the positions! So we specify more clauses of the form:

$$
\begin{array}{ll}
 & (\overline{x_{a1}} \vee \overline{x_{b1}}) \quad \text{// the number 1 cannot be in both positions a and b} \\
\wedge & (\overline{x_{a1}} \vee \overline{x_{c1}}) \quad \text{// the number 1 cannot be in both positions a and c} \\
\wedge & \cdots \\
\wedge & (\overline{x_{h9}} \vee \overline{x_{i9}}) \quad \text{// the number 9 cannot be in both positions h and i}
\end{array}
$$

Finally, to assure that all rows, columns, and diagonals add up to 15, we add clauses of the form: $(x_{a1} \wedge x_{b2} \Rightarrow \overline{x_{c3}})$. Meaning, if position a has a 1 and position b has a 2, then position c cannot have a 3. Now, this clause is not in disjunctive form, but it is easy enough to state it thus: $\overline{x_{a1}} \vee \overline{x_{b2}} \vee \overline{x_{c3}}$. Similarly, for each row, column, and diagonal we write out all combinations that will not add up to 15, and use them to produce clauses.

5. **(20 points)** Give each tile a name from the list [a,i] as in the previous question. Now take the variables and clauses from problem 4, excepting the clauses to assure sums of 15. This will ensure that each tile is placed in one unique place on the board. So far, so good. Now, they may or may not match up (with the colors).

   New Variables: Add 36 variables (for a total of 117) to represent a possible orientation of the tile in a certain position, written as follows $o_{\text{position orientation}}$ (i.e. $o_{a1}, o_{a2}, ..., o_{i4}$). So, if o_{a1} is true, it means that the tile in position a has the orientation 1 (e.g., facing north). Notice that this variable does not indicate what the tile is, it only uses board position and orientation.

   New Clauses: Once again there are a variety of clauses. First, every board position must have exactly one orientation.

$$
\begin{array}{lll}
 & (o_{a1} \vee o_{a2} \vee \cdots \vee o_{a4}) & \text{// the tile in a has at least one of the four orientations} \\
\wedge & (o_{b1} \vee o_{b2} \vee \cdots \vee o_{b4}) & \text{// the tile in b has at least one of the four orientations} \\
\wedge & \cdots & \\
\wedge & (o_{i1} \vee o_{i2} \vee \cdots \vee o_{i4}) & \text{// the tile in i has at least one of the four orientations} \\
\wedge & (\overline{o_{a1}} \vee \overline{o_{a2}}) & \text{// the tile in a cannot have both orientations 1 and 2} \\
\wedge & (\overline{o_{a1}} \vee \overline{o_{a3}}) & \text{// the tile in a cannot have both orientations 1 and 3} \\
\wedge & \cdots & \\
\wedge & (\overline{o_{i3}} \vee \overline{o_{i4}}) & \text{// the tile in i cannot have both orientations 3 and 4}
\end{array}
$$

   So far we have ensured that each tile is in exactly one board position and oriented in exactly one way. Now comes the tricky part. Take a combination of board positions, and for every combination of tiles that will not work for those positions, add a clause to assure that this combination should be excluded. Say, tile 1 will not work in board position a in orientation 2 if tile 2 is sitting in board position b in orientation 3 (from the colors of the ropes). We add a clause of the form:

$$x_{b2} \wedge o_{b3} \Rightarrow \neg(x_{a1} \wedge o_{a2})$$

   Meaning, if board position b has tile 2 sitting in orientation 3, then board position a cannot have tile 1 sitting in orientation 2. This can easily be restated as a disjunction:

$$\overline{x_{b2}} \vee \overline{o_{b3}} \vee \overline{x_{a1}} \vee \overline{o_{a2}}$$

   There are many more solutions. One approach is to have variables that are indexed by three subscripts, i.e., x_{a13} is true if tile 1 is placed in board position a with orientation 3. Another is to encode the rope colors directly in the description of the tiles.

6. **(30 points)** Once again there are many ways to solve this exercise. The variables could either represent the board positions or they could represent the tiles. Lets go with the latter formulation.

The variables are the nine tiles that had to go on the board. Their domains are a tuple of the possible positions on the board $\times$ the orientation of the tile, so for a 3 by 3 board, the initial domain size was 36. These variables had a constraint with each of the variables for tiles surrounding them that their patterns matched up, and with all other tiles that the same position was not occupied. This meant that every variable was constrained by every other.

Once the CSP was in place, it is a simple matter to implement the algorithms from the book, with the results for a 3×3 board summarized in the following table (all times are in seconds):

Search Method	Propagation Time (thinking)	Search Time (doing)	Total Time
Backtracking	0.0	0.25	0.25
Forward Propagation	0.188	0.047	0.235
AC3	0.078	0.0	0.078

As can be seen from the table, arc consistency runs the fastest with almost all of its time being spent in checking arc-consistency. As a result of the small size of the board, arc-consistency performs very well since it almost entirely solves the problem. Unfortunately this does not scale up well as a result of the fact that for larger boards, arc-consistency does not result in as close a path to the goal. For this search, forward propagation did not provide a large benefit over backtracking, since the propagation did little to shrink the search space, while taking a relatively long time to perform.