Nick Morris
Matthew Babcock

Integer & Nonlinear Programming
Midterm Exam

Overview:

Slitherlink puzzles are logical puzzles that have similar requirements as a travel salesman problem. Slitherlink puzzles consist of a collection of nodes with numbers centered within sets of 4 nodes. The main objective of a Slitherlink puzzle is to connect the nodes with a single closed loop that doesn't cross over itself. Additionally, the player must ensure that every number is bordered by the correct amount of arcs. The correct amount of arcs is dependent upon the number centered in the grid. For instance, if the number was a (3) there needs to be exactly three arcs bordering the corresponding grid. The objective of this paper is to formulate a generalized Integer Program that can solve any Slitherlink puzzle, and then apply it to solve the Slitherlink puzzles shown in Figure 1.

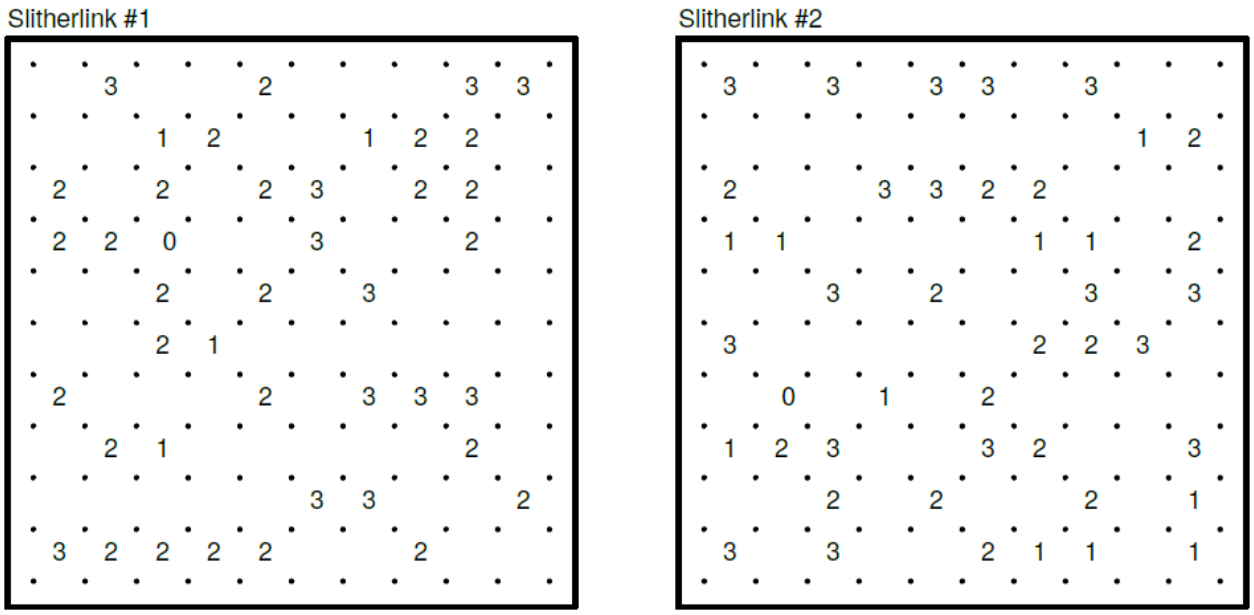

Figure 1: Slitherlink Puzzles

Methodology:

Base Model:

The slitherlink puzzle has similar properties of a traveling salesman problem and a network flow problem. The standard optimization formulations for these two types of problems were combined and adapted to create the following base model, an integer program. This model will return a solution that creates multiple closed loops, subtours that do not overlap, and satisfy the number value constraints in a slitherlink puzzle. This is the first step in finding the final solution which uses just one closed loop to satisfy all of the number values in a slitherlink puzzle.

The set of nodes correspond to the nodes seen in the puzzle, where each node is given a unique ID number. The set of bidirectional arcs correspond to all possible ways to draw adjacent links between nodes, this means that every pair of nodes have two arcs. The set of grids correspond to four nodes that create all grids with numbers in their center. The order of the four nodes in the indice of a grid do not matter for set G.

The Demand constraint is taken from the network flow formulation and adjusted to apply to the demand of a grid, as opposed to the demand of a node as in the original network flow formulation. This constraint states that the sum of all possible bidirectional arcs corresponding to a grid must be equal to the demand of that grid. The NetFlow constraint is taken directly from the network flow formulation without any adjustments made. This constraint states that the difference between all possible ingoing and outgoing arcs at a node must equal zero. The OneIn and OneOut constraints are taken from the traveling salesman formulation and were adjusted by becoming inequalities. This is because not all nodes have to be visited, unlike in the traveling salesman problem. These constraints state that the sum of all possible ingoing arcs at a node can be no more than one, and the sum of all possible outgoing arcs at a node can be no more than one. The OneWay constraint was added to facilitate the behavior of the NetFlow constraint, while making sure the Demand constraint was satisfied. If the OneWay constraint wasn't included, then the program would take advantage of the bidirectional arcs and use both directions of one arc to satisfy two demand units. This constraint states that up to one direction of a bidirectional arc can be used.

```
param n := 11;      # number of nodes per row/column

set N := 1..(n * n);      # nodes
set A within N cross N;   # bidirectional arcs
set G within N cross N cross N cross N;       # grids

param d{(i, j, k, l) in G};      # demand of a grid
param b{i in N} default 0;       # net flow at a node
```

```
var x{(i, j) in A} binary;       # do or don't use an arc

minimize Arcs: sum{(i, j) in A}(x[i, j]);     # minimize arcs used

s.t. Demand{(i, j, k, l) in G}: (x[i, j] + x[j, i]) + (x[l, k] + x[k, l]) +
                                (x[i, l] + x[l, i]) + (x[j, k] + x[k, j])
                                = d[i, j, k, l];
# satisfy the demand of all grids

s.t. NetFlow{j in N}: sum{(i, j) in A}(x[i, j]) - sum{(j, k) in A}(x[j, k]) = b[j];
# satisfy the netflow at all nodes

s.t. OneWay{(i, j) in A}: x[i, j] + x[j, i] <= 1;
# only one direction of an arc can be used at most

s.t. OneIn{j in N}: sum{(i, j) in A}(x[i, j]) <= 1;
# there can only be one inbound arc to a node at most

s.t. OneOut{j in N}: sum{(j, k) in A}(x[j, k]) <= 1;
# there can only be one outbound arc to a node at most
```

Removing Subtours:

The next step in the procedure for solving a slitherlink puzzle is to remove subtours. The parameter s, sets SF and S, and Subtours constraint shown below are added to the base model to remove subtours. The set of subtours (SF) is a set of unique ID numbers for each subtour. The set of subtour arcs (S) consist of all arcs for each subtour. The Subtours constraint was taken from the traveling salesman formulation and adjusted to account for bidirectional arcs. This constraint states that at least one of bidirectional arcs in each subtour cannot be used.

This step in the process requires a procedure for extracting the set of subtours from the current solution. This was handled by creating a function in R that uses a double while loop to read the nodes of a solution and return the sets of subtour arcs that are to be imported into the data file. Once these sets of subtours are imported into the data file, the program can be resolved without those subtours being used. This step is iterative because the next solution may use a different set of subtours, so the data file may grow significantly until enough subtours have been added such that the program uses one closed loop to solve the puzzle properly.

```
param s;           # number of subtours

set SF := 1..s;    # subtours
set S{SF} within N cross N;      # subtour arcs

s.t. Subtours{k in SF}: sum{(i, j) in S[k]}(x[i, j] + x[j, i]) <= card(S[k]) - 1;
# break known subtours
```

The following steps describe how the R function works for finding and returning the subtours.

1. Let the beginning of the current solution table look like the following:

```
InNode OutNode
1      12
2      1
3      2
4      15
5      4
.      .
.      .
.      .
```

2. Create an empty vector → row
3. Store the row index of the first row into the empty vector → row
4. Store the InNode of the first row → start.node
5. Store the OutNode of the first row → next.node
6. Look up the value of next.node in the column InNode, and store the row index of the match → i
7. Append the value of i to the vector → row
8. Store the OutNode of row i → next.node
9. Evaluate if next.node == start.node
    a. If TRUE then a subtour was found
        i. Remove all rows from the current solution table with the row indices in the vector row and store these rows as a separate table → Subtour-j
    b. If FALSE then a subtour hasn't been found yet
        i. Go back to step 6
10. Evaluate if the updated current solution table still has more than zero rows
    a. If TRUE then there is at least one more subtour left to find
        i. Go back to step 2
    b. If FALSE then there are no more subtours left to find
        i. Format all Subtour-j tables in ampl syntax
        ii. Print the Subtour-j tables
11. End

Additional Cuts:

There were four general rules applied to create additional cuts.

Referencing Figure 2, the rule in the top left states that if 3's are adjacent to each other then the final solution will have one line drawn between these grids and a line drawn on each end of the grids. If these lines were not drawn, then a single loop would not be possible.

The rule in the top right states that if a number is in a corner then the final solution will have no lines in the corner for a 0 and 1, two lines leading up to a 2 (where another two lines may or may not wrap around the corner), and two lines wrapping around the corner for a 3. If these lines were not drawn, then satisfying these grid demands would not be possible.

The rule in the bottom left states that if a pair of 3's are diagonally adjacent to each other, then two lines must make a corner around each 3, with the two corners facing each other. If these lines were not drawn, then a single loop would not be possible.

The rule in the bottom right states that if a 3 and a 0 are diagonally adjacent to each other, then two lines must make a corner around the 3, facing away from the 0. If these lines were not drawn, then satisfying both of these grid demands would not be possible.

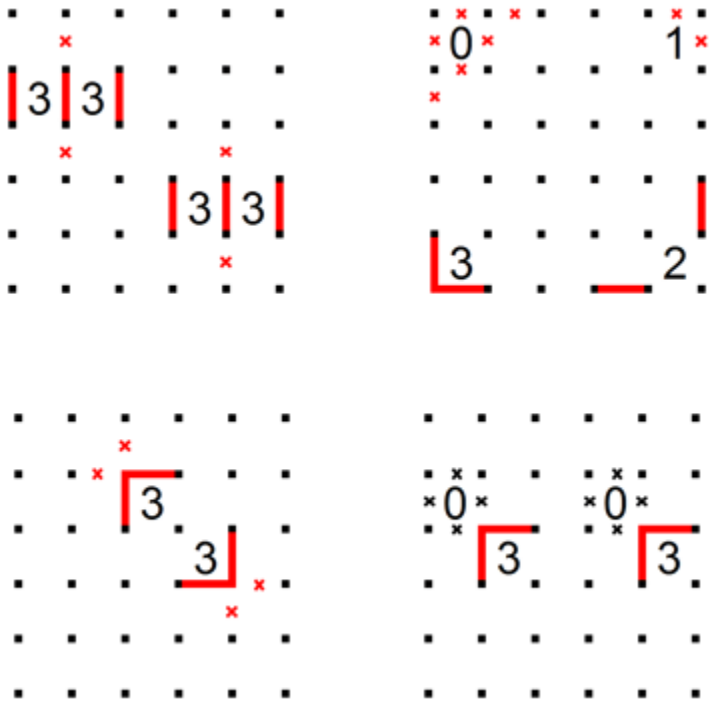

Figure 2: Slitherlink Starting Rules

The additional cuts can be included with the base model through a set of known arcs that are in the final solution. The Cuts constraint simply states that one of the bidirectional arcs in the set K must be used. The addition of this set and constraint is expected to reduce the total number of simplex iterations and branch and bound nodes to evaluate, which will be covered for the two slitherlink examples in the discussion section.

```
set K within N cross N;   # known arcs

s.t. Cuts{(i, j) in K}: x[i, j] + x[j, i] = 1;
# satisfy all known arcs
```

Final Model:

The following is the final model, which is our integer program to solve any slitherlink puzzle.

```
param n := 11;      # number of nodes per row/column
param s;            # number of subtours

set N := 1..(n * n);      # nodes
set A within N cross N;   # arcs
set G within N cross N cross N cross N;       # grids
set K within N cross N;          # known arcs
set SF := 1..s;           # subtours
set S{SF} within N cross N;      # subtour arcs

param d{(i, j, k, l) in G};      # demand of a grid
param b{i in N} default 0;       # net flow at a node

var x{(i, j) in A} binary;       # do or don't use an arc

minimize Arcs: sum{(i, j) in A}(x[i, j]);     # minimize arcs used

s.t. Demand{(i, j, k, l) in G}: (x[i, j] + x[j, i]) + (x[l, k] + x[k, l]) +
                                (x[i, l] + x[l, i]) + (x[j, k] + x[k, j])
                                = d[i, j, k, l];
# satisfy the demand of all grids

s.t. NetFlow{j in N}: sum{(i, j) in A}(x[i, j]) - sum{(j, k) in A}(x[j, k]) = b[j];
# satisfy the netflow at all nodes

s.t. OneWay{(i, j) in A}: x[i, j] + x[j, i] <= 1;
# only one direction of an arc can be used at most

s.t. OneIn{j in N}: sum{(i, j) in A}(x[i, j]) <= 1;
# there can only be one inbound arc to a node at most

s.t. OneOut{j in N}: sum{(j, k) in A}(x[j, k]) <= 1;
# there can only be one outbound arc to a node at most

s.t. Cuts{(i, j) in K}: x[i, j] + x[j, i] = 1;
# satisfy all known arcs

s.t. Subtours{k in SF}: sum{(i, j) in S[k]}(x[i, j] + x[j, i]) <= card(S[k]) - 1;
# break known subtours
```

Discussion:

Slitherlink #1 Solution:

The following figure is a feasible solution to the slitherlink #1 puzzle from the integer program. The black numbers on grey grids indicate the demand of a grid. The white numbers on red grids, and the black numbers on white grids are the node ID numbers. The red grids show the slitherlink path that satisfies the demand of all grey grids.

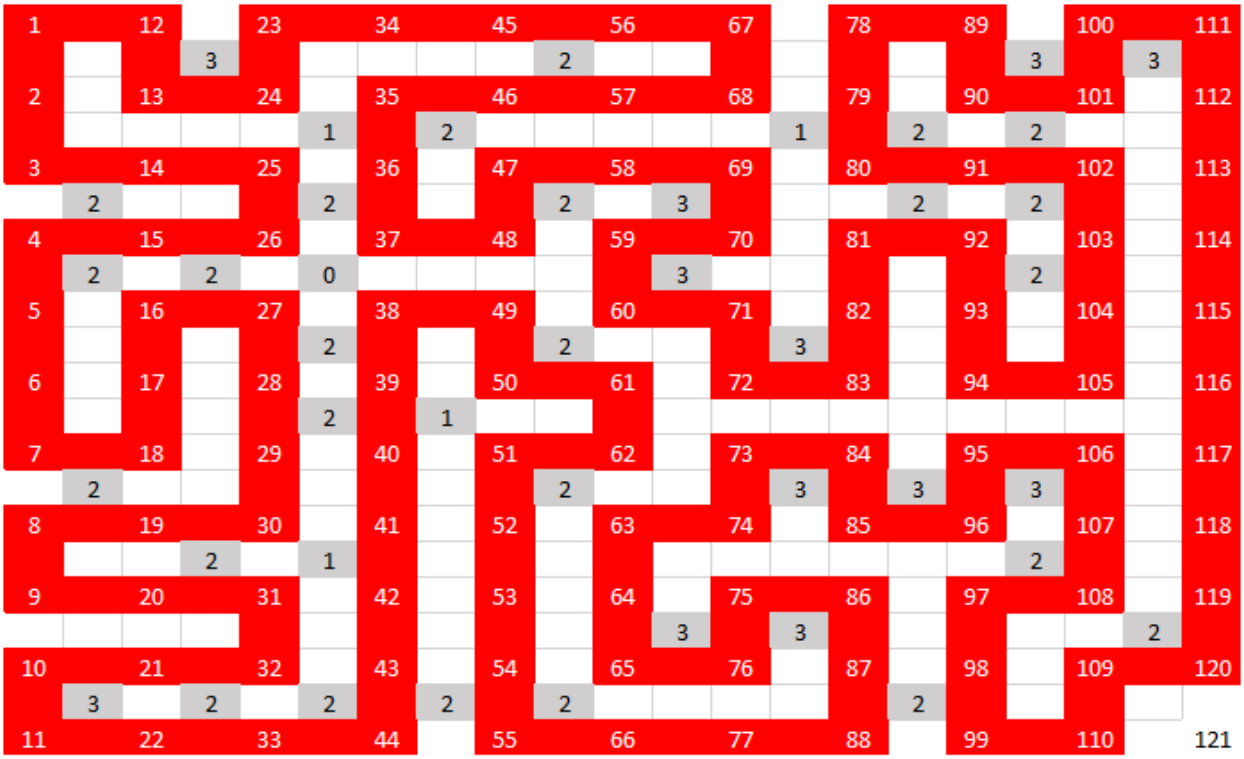

Figure 3: Feasible Solution to Slitherlink Puzzle #1

Slitherlink #1 Without Additional Cuts:

The cplex performance summary is shown below in Table 1. The solution time shows that our program can solve this puzzle in about 3 seconds. The first run had a search tree size of 224 nodes, and then the remaining runs were able to find the optimal solution at the root node. This means that for runs 2 to 6, the continuous relaxation of the problem was solved and returned an integer solution. The number of simplex iterations show a 17% drop after run 1 and then levels off. Ultimately, 15 subtours were identified to solve the problem with our program.

Table 1: Performance for Slitherlink Puzzle #1 without Additional Cuts

Run ID	Solution Time (sec)	B&B Nodes	Simplex Iterations	Subtours
1	0.84	224	7423	4
2	0.42	0	1253	3
3	0.41	0	1520	4
4	0.53	0	1483	2
5	0.48	0	1405	2
6	0.48	0	1215	0
Total	3.16	224	14299	15

The time series of the MIP gap for each of the 6 runs is plotted below in Figure 4. The plot shows that for each run, the MIP gap decreases slowly up until a drop at the end. Notice that as more subtours are identified, the the time series length decreases. Run 1 was the only run which couldn't achieve a zero gap.

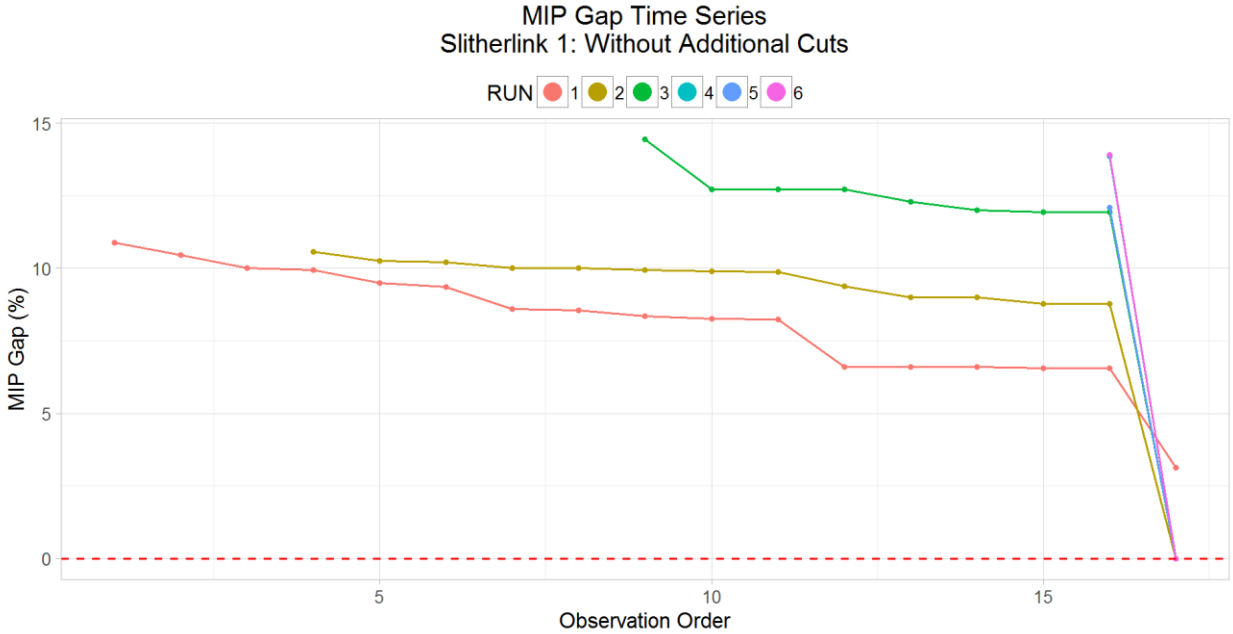

Figure 4: MIP Gap Time Series for Slitherlink #1 Without Additional Cuts

Slitherlink #1 With Additional Cuts:

The cplex performance summary is shown below in Table 2. The solution time shows that our program can solve this puzzle within one second. All runs except for run 3, were able to find the optimal solution at the root node. The number of simplex iterations show a steady decline as more subtours are found. Ultimately, 11 subtours needed to be identified to solve the problem properly with our program.

Table 2: Performance for Slitherlink Puzzle #1 with Additional Cuts

Run ID	Solution Time (sec)	B&B Nodes	Simplex Iterations	Subtours
1	0.2	0	827	4
2	0.2	0	568	4
3	0.14	32	360	3
4	0.12	0	185	0
Total	0.66	32	1940	11

The time series of the MIP gap for each of the 4 runs is plotted below in Figure 5. The plot shows that for each run, the MIP gap decreases steadily. Run 3 was the only run which couldn't achieve a zero gap.

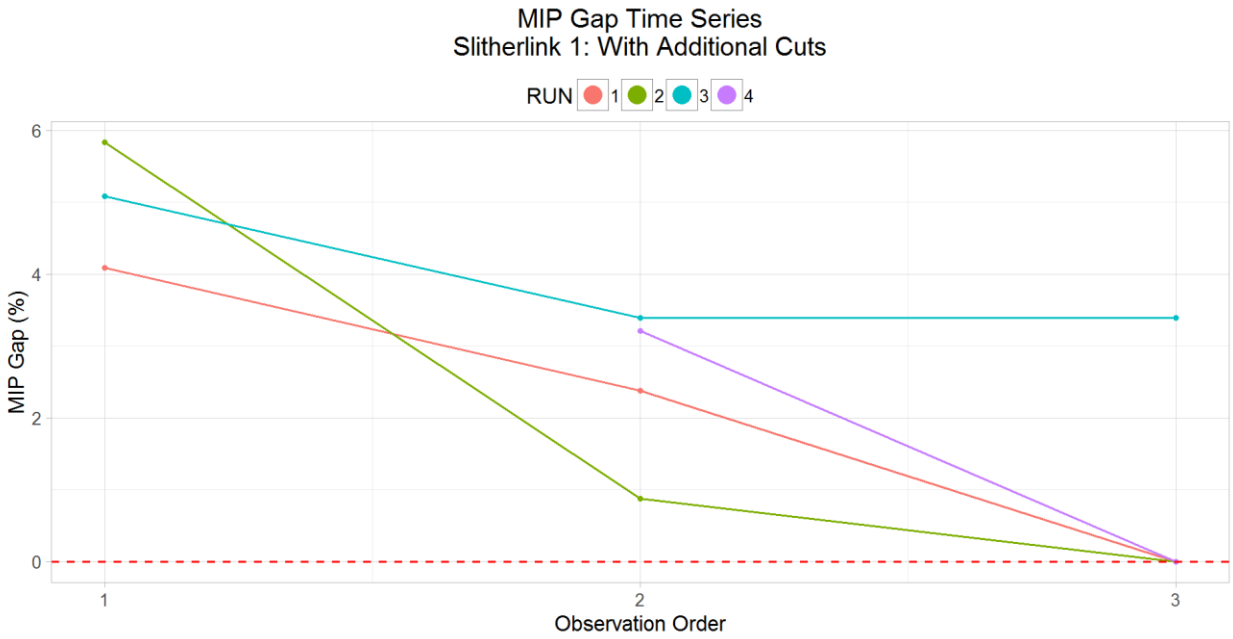

Figure 5: MIP Gap Time Series for Slitherlink #1 With Additional Cuts

Justification of Slitherlink #1 Cuts:

Table 3 below shows that the addition of cuts made a significant difference for our model performance with large reductions in solution time, simplex iterations, and search tree size. The number of subtours required to find decreased which resulted in less times the program needed to be run. The use of cuts didn't impact the final MIP gap, both approaches achieved a solution with a 0% gap. Overall the addition of cuts improved performance.

Table 3: Comparison of Performance for Slitherlink #1 Puzzle

Approach	Solution Time (sec)	B&B Nodes	Simplex Iterations	Subtours
Total Without Cuts	3.16	224	14299	15
Total With Cuts	0.66	32	1940	11
Difference	-2.5	-192	-12359	-4

Slitherlink #2 Solution:

The following figure is a feasible solution to the slitherlink #2 puzzle from the integer program.

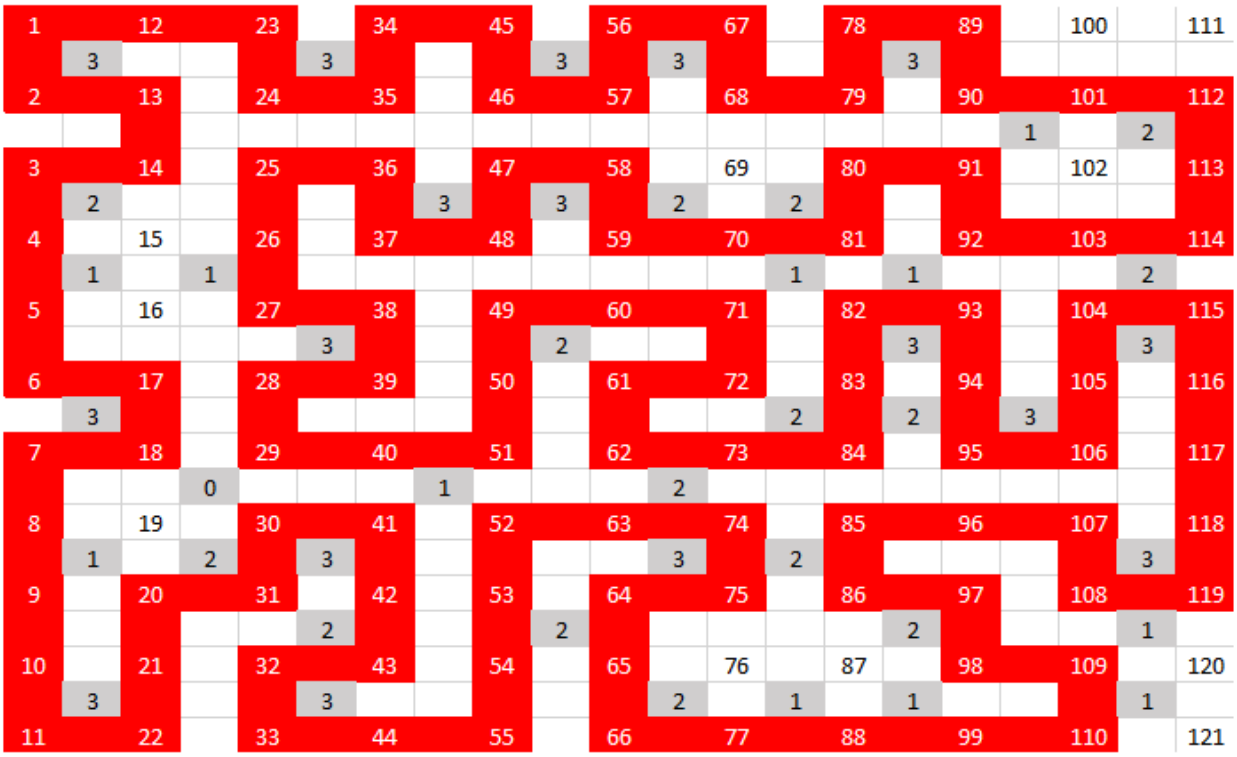

Figure 6: Feasible Solution to Slitherlink Puzzle #2

Slitherlink #2 Without Additional Cuts:

The cplex performance summary is shown below in Table 4. The number of branch and bound nodes explored in the first run was found to be 0, meaning the solution was at the root node. There were 740 simplex iterations with a solution time of 0.11 seconds taken in order to find two subtours. Breaking these subtours in run 2 presented a solution time of 0.31 seconds across 513 branch and bound nodes, and 10584 iterations. Run 2 yielded a single tour, therefore no subtours.

Table 4: Performance for Slitherlink Puzzle #2 without Additional Cuts

Run ID	Solution Time (sec)	B&B Nodes	Simplex Iterations	Subtours
1	0.11	0	740	2
2	0.31	513	10584	0
Total	0.42	513	11324	2

The time series of the MIP gap for the 2 runs is plotted below in Figure 7. The plot shows that for each run, the MIP gap was unchanged. Run 2 wasn't able to achieve a zero gap, resulting in a final solution with a 3.64% gap. This is likely due to breaking the two subtours, which made the program more difficult to solve, as shown by the large increase in simplex iterations.

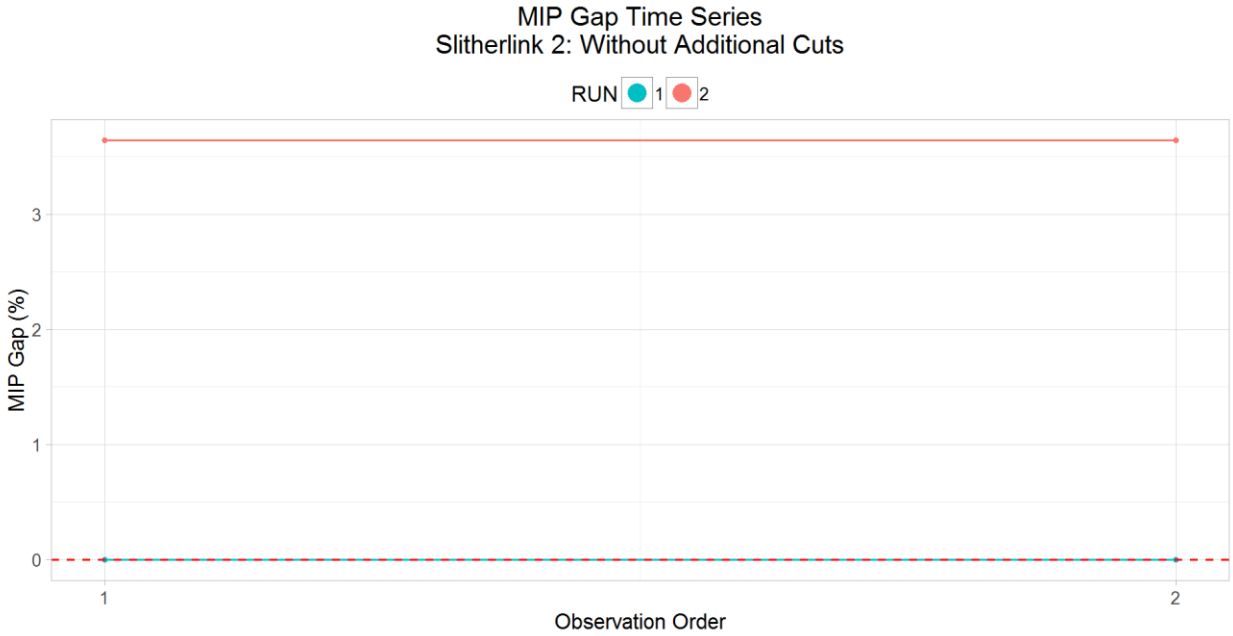

Figure 7: MIP Gap Time Series for Slitherlink #2 Without Additional Cuts

Slitherlink #2 with Additional Cuts:

The cplex performance summary is shown below in Table 5. Adding cuts yielded two runs with large improvements in total solving time, branch-and-bound nodes, and number of simplex iterations. As shown in Table 5, the first run took 0.22 seconds to explore 0 branch-and-bound nodes for 308 iterations to find 3 subtours. The second run took 0.14 seconds to explore 0 branch-and-bound nodes for 380 iterations to ensure a single tour.

Table 5: Performance for Slitherlink Puzzle #2 with Additional Cuts

Run ID	Solution Time (sec)	B&B Nodes	Simplex Iterations	Subtours
1	0.22	0	308	3
2	0.14	0	380	0
Total	0.36	0	688	3

The time series of the MIP gap for the 2 runs is plotted below in Figure 8. The plot shows that for each run, the MIP gap was steadily decreasing to a zero gap except for run 1 which started off at a large gap over 50%.

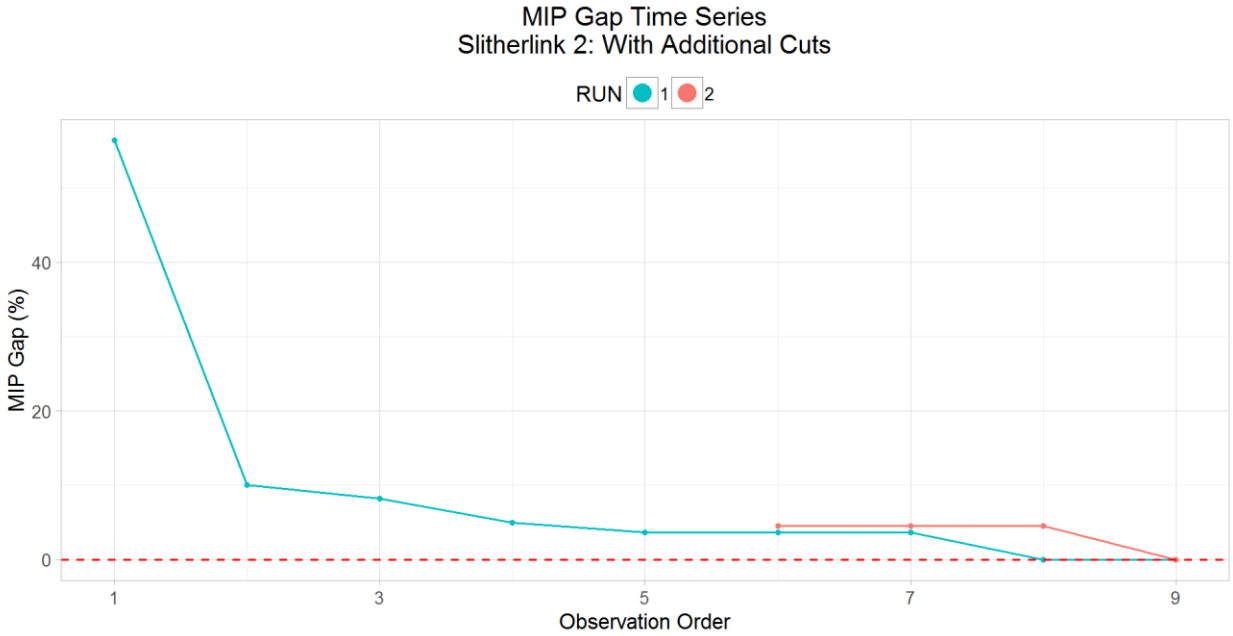

Figure 8: MIP Gap Time Series for Slitherlink #2 With Additional Cuts

Justification of Slitherlink #2 Cuts:

The comparison of the two approaches is shown in Table 6. Adding the cuts reduced the solution time by 0.06 seconds. Adding cuts also found its solutions at the incumbent which is 513 less branch-and-bound nodes then the solution without cuts. The number of simplex iterations reduced by 10636. The number of subtours found by the total cuts increased by 1 although it was solved in the same number of runs. Adding cuts also yield a final solution with a 0% gap, whereas a gap of 3.64% is achieved without the use of additional cuts. Overall the addition of cuts improved performance.

Table 6: Comparison of Performance for Slitherlink #2 Puzzle

Approach	Solution Time (sec)	B&B Nodes	Simplex Iterations	Subtours
Total Without Cuts	0.42	513	11324	2
Total With Cuts	0.36	0	688	3
Difference	-0.06	-513	-10636	1

Conclusion:

We have engineered an optimization based approach to solving the slitherlink puzzle. Examples of how the slitherlink puzzle can be applied to real world problems, hence how our model is applicable to real world problem solving:

- Terrain negotiation (Hiking around impassable terrain)
- Traveling in City Traffic
- GPS traveling around fixed obstacles

Limitations:

We have a reactive procedure to subtours as opposed to a proactive procedure. This means that we could be re-solving this problem many times until it achieves a single loop solution. Another limitation is that we have constraints from the standard network flow problem but failed to maintain total unimodularity. Total unimodularity would allow our binary variable to become a nonnegative real variable and thus our model would become a linear program that returns integer solutions.

Areas for Improvement:

The first improvement that could be made to our model would be to generate all subtours initially and then move the subtour constraint to the objective function through Lagrangian relaxation. Another improvement is to modify our constraints to achieve total unimodularity and make this a linear program. The next improvement that could be done is to automate data exchange between ampl and R such that the user only needs to run one command to solve the problem. Finally, implementing more of the starting rules as additional cuts. For example, having a set K2 of known no-arcs such that x[i, j] + x[j, i] = 0 for every [i, j] in K2.