Implementation of Eight Neighbor Grids

Eight neighbor grids are represented by a model that uses a Graph class and a Vertex class.

Vertex Class
The Vertex class represents nodes on the eight neighbor grid and stores their g, h, and f-values. Vertex objects are identified by a unique key in the form “x|y”. The class also maintains a neighbors dictionary with neighboring vertices as the key and that edges weight as the value.

Graph Class
The graph class maintains a list of all of the participating vertices, the starting and goal vertices and a dictionary that keeps track of which cells are blocked and unblocked.

These grids interact with the front end visualization by providing a Graph object that represents the eight neighbor grid with a start and goal node. The path is generated later by the A* or Theta* algorithm class

Implementation of Algorithms

A* and Theta* are implemented as classes with a main method closely following their respective pseudocode.

AStar Class
The AStar class is interacted with through its main method. It accepts a graph object as a parameter and then when the main method is run [*AStar(graph).main()*] it returns the shortest path as a list of nodes from start node to goal node. AStar implements *main, g, h, shortest_path, UpdateVertex.*

shortest_path(end_node)

- *Given the final node after running A* traces the path to that node through its parents and returns a reversed list.*

Note: All other methods are implemented exactly as described in the project description

Problem 1g - Optimization of A*/Theta*

Removing Redundant Code

The first step in optimization was removing redundant code. Initially we used an implementation that traced the path from a node to the start through parents each time g was called. After realizing the pseudocode properly updated a Vertex's g-val we adapted our code to take advantage of this. This provided significant reductions in runtime of the code but is not discussed in detail here because the implementation represented a misunderstanding of the pseudocode and not a true optimization.

G-Attribute Access

In the initial implementation of A*/Theta* g-values are set through a custom setter method called *vertex*.setG(). This method was used in order to update the f-value automatically whenever the g-value was set in the code. F-values however do not need to be stored updated and only need to be updated before being added to the heap. Switching to dot attribute access, not including the time it takes to update f-value, saves about 20% of the time. This setting g-value operation is done twice per loop so this saves a fair amount of time each loop. Additionally, by unwrapping the updating of f-values we save unnecessary updates and only update when necessary. This is a small runtime optimization and has the downside of complicating the codebase slightly. It is less robust in terms of maintaining the codebase.

Average setG Runtime	Average dotG Runtime	Percent Change
0.01023563121	0.00857510101	19.36%

Note: Average runtimes were calculated using the timeit module and are the result of 1,000,000 tests

Closed Data Structure

The initial implementation of the closed data structure is a Python list. The operations performed on this list are append and search. Append is performed in O(1) time to a list which is optimal but searching the list is Average Case O(n). There are ways to maintain the closed list that can perform better than this. Python dictionaries are implemented as hash tables so they have an Amortized Worst Case lookup of O(1).

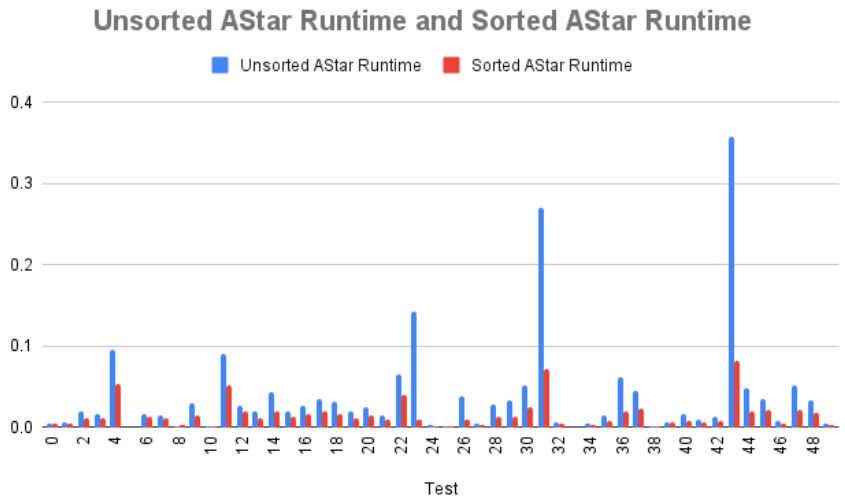
Dictionary(Hashtable) Implementation

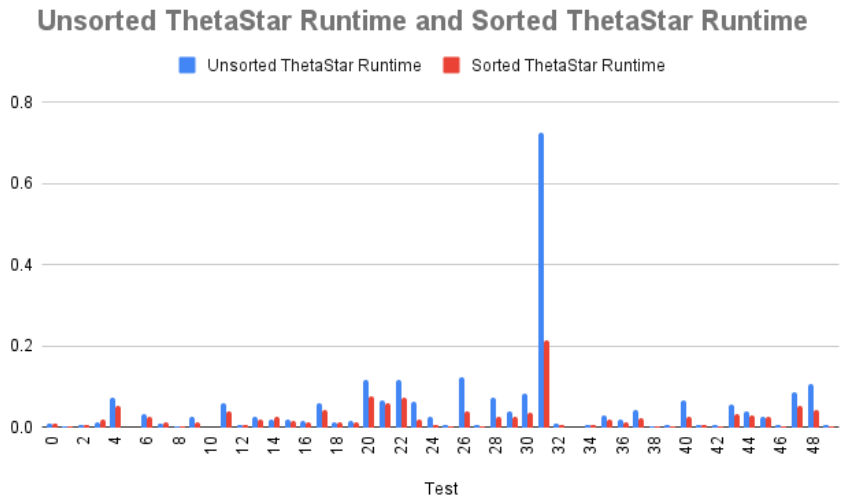
One possible implementation of the closed list is with a dictionary. We can search a dictionary for the key of a vertex with an average case O(1) but a worse case of O(n) so this is only slightly better. By initializing a dictionary with a key for every vertex and setting values to 0 or 1 (in closed or not in closed) we can lower the search time for within the algorithm to O(1) but this

would require a setup or tear down where all vertex key values are set to 0. The added memory usage and setup/tear down costs made this not an optimal solution for this implementation.

Sorted List w Binary Search Implementation

A better implementation is to store the list as vertex keys in sorted order and search the closed list using binary search. Binary Search is implemented in its iterative version which has O(1) memory complexity and O(logn) time complexity for search. Keys are added to a sorted list in O(log n) using the python bisect library with a python list. The runtime improvements of the algorithm are significant and are shown in the trials below.





Problem 1h

Experimental Design

Data for the analysis of A*/Theta* was collected by generating 200 random eight neighbor grids and running the A* algorithm and Theta* algorithm on each of the test grids. For each grid we observed the A* Runtime, Theta* Runtime, A* Path Length and Theta* Path Length. Grids are generated using the generatetests.py file and randomness is enforced with the built in random module.

Analysis of Path Lengths/Runtimes

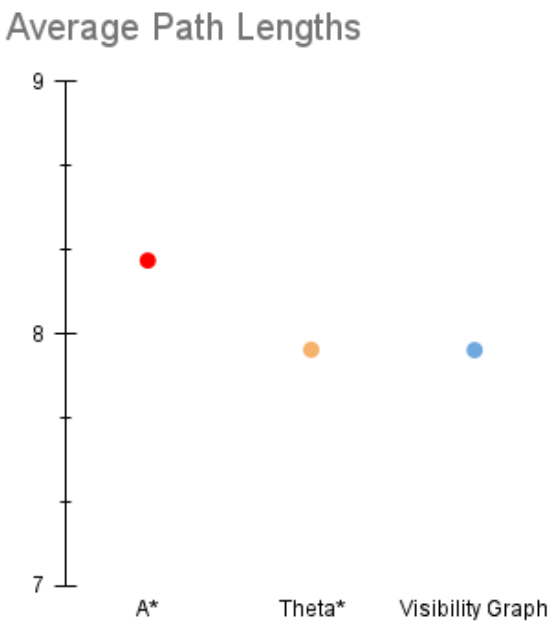
Averages			
AStar Runtime	AStar Path Length	ThetaStar Runtime	ThetaStar Path Length
0.01740716242	41.77888273	0.03091763402	39.86143948

A* outperformed Theta* by 43.7% in terms of runtime. In terms of path lengths Theta* averaged about 4% shorter path lengths. The increase in runtime due to the amount of LineOfSight checks that are performed in Theta*. Every time that Theta* runs UpdateVertex and has the initial overhead of performing a LineOfSight check to choose its path. Optimization of Theta* would have to be largely around how LineOfSight checks are performed. The LineOfSight function used is relatively simple and cannot likely be optimized much however. The current LineOfSight implementation runs through conditional logic with relatively few jumps, does simple calculations with numbers generated through array access, and calls the cell is blocked method. Outside of micro optimizations the only thing that could be changed is the implementation of cellisblocked. CellIsBlocked mainly reads dictionary values (O(1) operation) and performs simple conditional checks. In conclusion there doesn't appear to be much room for optimization in the current LineOfSight implementation in order to make significant progress towards closing the gap between A* and Theta* a different LineOfSight method would be needed. The benefit that Theta* provides is a marginal decrease in path length so any discussion around using it would have to be around whether the 4% increase is significant enough at scale or whether it is important to have paths appear smooth (such as in video game AI or robotics).

Use of Different H-Values

H-Values used in Theta* and A* in terms of runtime are essentially the same. They are both values that can be calculated in O(1) time. The differences in the h-values appear when looking at their outputs. A* Equation 1 consistently outputs a value larger than Theta* c() over 1,000,000 tests with random vertex coordinates. On average the distance between these two h-value equations is about 2.54. This is significant because A* Equation 1 is consistent and admissible meaning it is never greater than the actual distance. If A* Equation 1 is consistent and admissible and always generates an output that is 2.54 higher than Theta* c() it will always be closer to the true distance. A heuristic that is closer to the true distance is always a better heuristic as long as it never over estimates. Therefore the use of different heuristics results in a significant advantage for the A* algorithm.

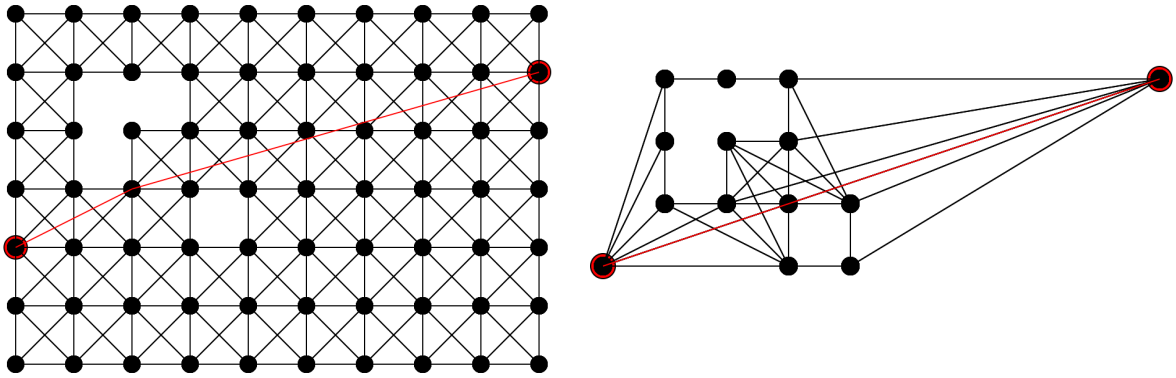
Problem 1i



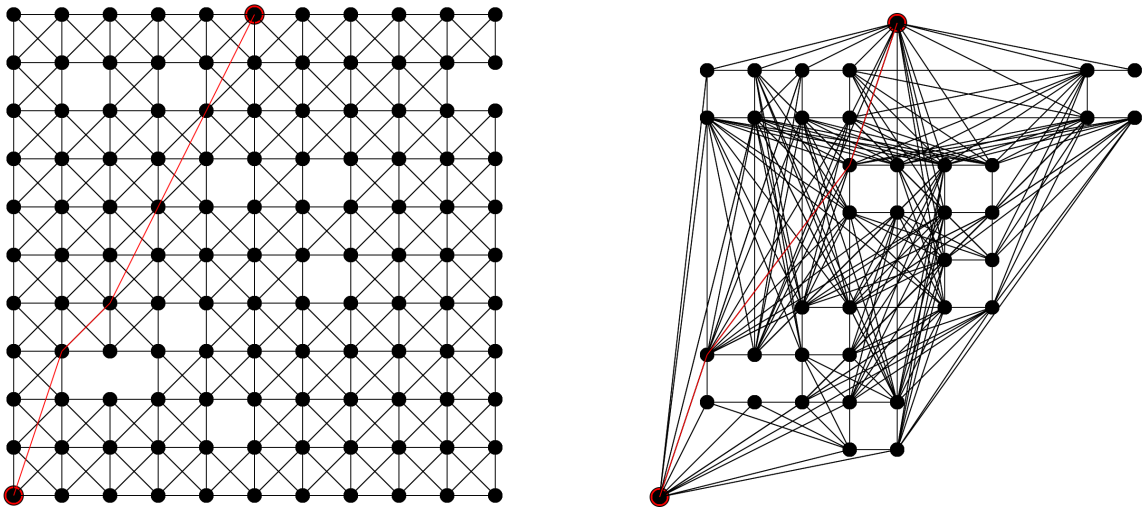
True Shortest Paths

Theta* requires vertex parents to be a neighbor that passes the LineOfSight test or the parent of a neighbor that passes the line of sight test. These are not always the shortest paths. Visibility graphs are guaranteed to generate the shortest paths so they can be used in test cases as the Expected answer while Theta* is the Actual result. By implementing a series of random walks on grids 10x10 we can quickly locate cases where the true shortest path is not found by Theta*. The graph pictured about shows the experimental results of 50 cases where the true path length is shorter than the Theta* path length. The average path lengths in the case where Theta* is not the shortest path are listed below.

A*	Theta*	Visibility Graph
8.290336213	7.93662637	7.935061797



Above is a manually generated example of when Theta* does not find the true shortest path.



Pictured above are two randomly generated cases where Theta* fails to find the shortest path.

Implementation of Visibility Graphs

Visibility graphs are implemented in the program using a simplified version of the Graph class. They share a simplified set of methods and attributes with the Graph class. The vis.py file can be called from the command line with a graph filename and it will display the VisibilityGraph for the graph it is given. More details on this are given in the readme.

Problem 2: Adversarial Search

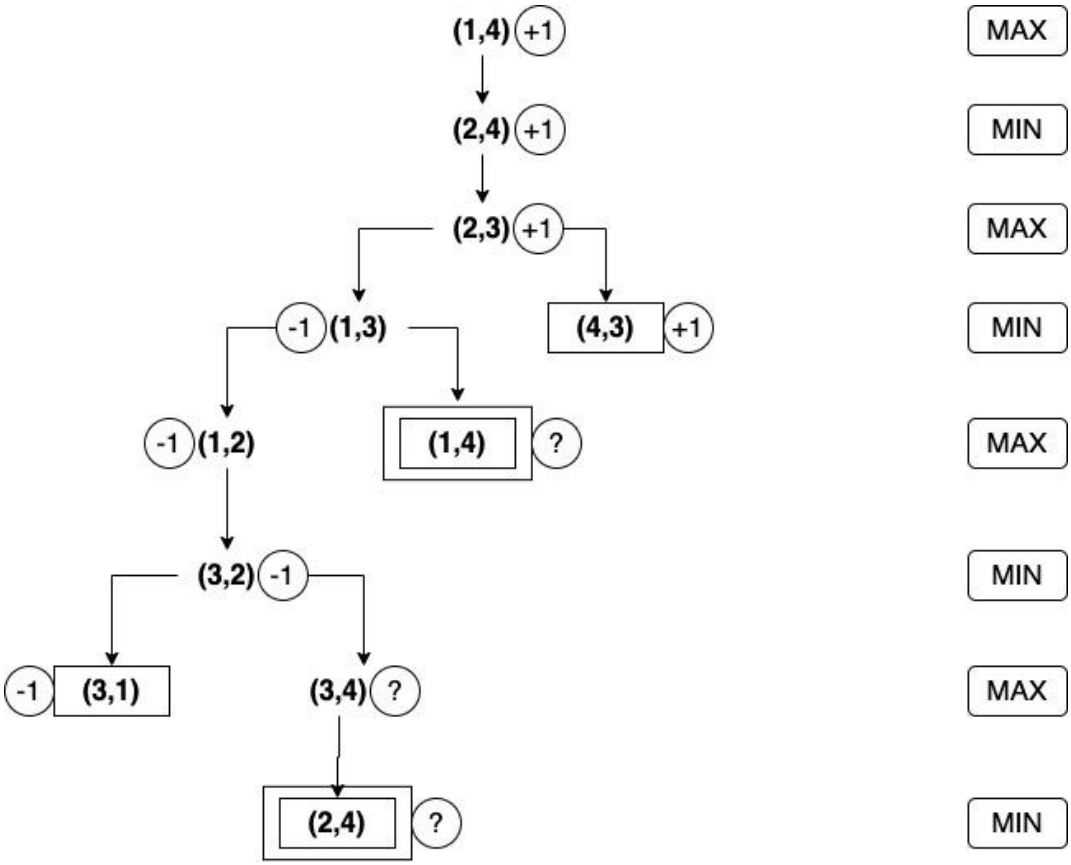


Figure 1: The game tree for the game described in Problem 2. Loop states are denoted with a double square. Terminal states are denoted with single squares. States are represented in tuples in the form (S_A, S_B). Minimax values are assigned as explained in part b.

B. In both loop states, the player whose turn it is stays the same. For example the state (2,4) will always be played by the MIN player. Therefore there is no utility benefit to choosing a loop state. The minimax algorithm should instead ignore the loop states and always pick other states instead.

C. The standard minimax algorithm would fail because the game tree is **infinitely deep.** A modified algorithm could solve this problem by recognizing when a loop state is about to start and treating it as a terminal state.

This can be achieved by modifying the *terminal-state, utility, and min/max procedures.*

Terminal-State: Maintain a list of states that have been visited. Modify the Terminal Test to consider already visited states as terminals instead of allowing them to loop.
Utility Function: If a state is a repeated state, give it a value of "?" (or any distinct value from +1 | -1).

Min/Max Functions: Modify min/max functions to recognize "?"'s and only select them when they are the only option. In other words "?"s are treated as the lowest possible value by the max function and the highest possible value by the min function.

D. For a game over 3 squares, A will move first. B will have no choice but to step over A, and so will win the game.
For a game over 5 squares, A moves first and B moves second. A will be in the 3rd square from its start, and B will be 2nd from its own start. When A moves again, B's only next option is to step over A once more, leaving B one square away from its goal, while A needs two moves to win. In the state just before this one, the game's state was that of the 3-square state. It is essentially the same, since in either case B will be closer to its goal than A is.
In an odd-n game, after one move of A and one move of B, the odd-n game is essentially reduced to the beginning state of one odd-n game down from it. This can go down all the way to the size three game, which B is already confirmed to win. Even in a larger size-n game, if n is odd, at the stage where B crosses over A, B will have one less square to traverse to reach its goal than A does. No matter how many moves A will make, B will still win, in any odd-n game.
A will win in even-n games, because after A's initial move, B will essentially be starting in an odd-n game. As seen before, the one who starts an odd-n game will also lose it.

Problem 3 - Local Search

A. Hill climbing is effective at problems looking to quickly find local minima or maxima. If finding local maxima or minima is sufficient then hill climbing will find solutions faster than simulated annealing. Hill climbing by definition never makes downhill moves so it is impossible for it to be complete because without the randomness component it will always terminate if it finds local maxima or minima.
B. Random search can be useful for problems where the objective function is not smooth. If the objective function isn't smooth then it can be hard to calculate the gradient. The gradient is used by hill climbing to converge on a solution.
C. Simulated annealing is useful when value functions are smooth and contain a large amount of local maxima and minima. Simulated annealing is useful for quickly finding global maxima or minima by incorporating enough exploration early on in the search to avoid getting stuck in locals while not overshooting global maxima or minima.
D. Since simulated annealing terminates when the temperature schedule completes there is no guarantee that it terminates at a maxima or minima. After the termination of simulated annealing we can pass the current state to a hill climbing algorithm to ensure that the answer is a maxima or minima.
E. In the process for simulated annealing, after picking a random move the algorithm evaluates the move to see if it improves the situation. Currently the test to see if the move improves the situation is just a measure of whether that specific move is greater than the current position. Instead, if there is more memory, simulated annealing can be modified to use iterative depth first search to evaluate whether or not the randomly selected move is good or bad. By using the extra memory to look a few moves ahead the algorithm can better decide if a move is good or bad and converge on a solution faster.

Problem 4

A. The set of variables for the Sudoku Constraint Satisfaction Problem are all the cells on the Sudoku board. These can be divided into the cells which are empty when the board is initialized, and the cells which are filled at the time of the board's initialization. Each empty cell has a possible value of the numbers 1-9. The filled cells, on the other hand, will only have one value, the value they are initially filled with. The constraints for this problem require that every cell in a row has a unique non-repeating number, that every cell in a column has a unique non-repeating number, and that every square of 9 cells (of which there are also 9) is made of unique non-repeating numbers.

B. States for Sudoku would involve any amount of the empty cells being filled, with any of the numbers 1-9. States will include both boards that satisfy Sudoku constraints, and boards that do not. The initial state would be whatever the partially randomly filled board is, with some cells empty and some cells filled. Actions would be to fill in a cell with a value. The successor function would check the numbers which can go in a certain cell, based on the numbers in that cell's units (the cell's same column, row, or square). The goal test would require for every unit to have a different number from 1-9 with no repetitions, across the whole Sudoku board. As for the path cost, each cell needs every possible value to be considered. These would be the values which haven't already been used in any of the cell's same units.

The minimum value heuristic is better here, since the most constrained variables are easier to assign values to. As the most constrained variables get assigned values, the other variables on the board will also gain more constraints, until they too are considered for a value. The degree heuristic, on the other hand, may prematurely force the problem into a dead end.

The branching factor depends on the number of possible values per cell, for each empty cell in the Sudoku board.

The state space would contain every solution, even those in violation of Sudoku rules. This would mean the state space is as large as 9 values times the number of empty cells, for then three are no restriction on values in empty cells. The solution depth, on the other hand, cannot be deeper than the number of empty cells within the Sudoku board, because cells cannot be "over-filled". Likewise, the maximum depth cannot exceed the number of empty cells on the Sudoku board.

C. The difference between an "easy" Sudoku puzzle and a "hard" Sudoku puzzle are the decisions each requires an algorithm to make. In an "easier" Sudoku puzzle, it is more obvious which choice of value for a given variable will lead to a dead end. However, in a "hard" Sudoku puzzle, it's possible to go very far down a given solution path and not realize that that path will later only have options for Sudoku solutions in violation of the Sudoku rules.

D.

for each empty Sudoku cell/variable:

    fill with a random number from 1 to 9

Sudoku Solver Function (takes a random Sudoku cell as input**):**

```
    while true:
        if current + neighbors (above, below, left, right) comply with Sudoku constraints:
            break
        choose the most constrained neighbor (above, below, left, or right) of current cell
        if current cell's constraints > neighbor's constraints:
            change current cell's value to fit Sudoku constraints
        current cell = neighbor
```

[alt—iterate over possible values for a given cell, call itself w each different value, check if the current assignment results in success or not, and return any which are a successful assignment]

The function starts out by populating the empty cells of the Sudoku board with random numbers in the range 1-9, the possible values for variables of a Sudoku game. It then chooses a random Sudoku cell and loops over it and its neighbors. If the current Sudoku cell and the cells above, below, to the left, and to the right comply with Sudoku constraints, then the loop ends. Otherwise, the upper, lower, left, or right neighbor with the most constraints is chosen. If the current cell still has more constraints than the neighbor, then the current cell's value is what is changed. At the end, the neighbor becomes the current cell, and the loop repeats.

An actual walkSAT algorithm would be better at finding a solution or end to a problem faster than incremental search; however, it is not guaranteed to find a solution to every problem, since it is not complete. It would probably not be the best idea to use it for easy Sudoku problems, when an incremental search algorithm would likely not be too slow yet, while harder problems may take longer. However, it may not always find a solution in those harder problems.

5. (putting screenshots from LaTeX here)
https://www.overleaf.com/2266869829byzzwqrwhjnr

s = Superman
b = Batman
w = Wonder Woman

U_x = x is upset
A_x = x is alone
D_x = x is defeated
K_x = x has kryptonite

$C_{x,y}$ = x and y are collaborating or else on the same side.
Additionally, when x and y are collaborating or on the same side, it is also true that

$$C_{x,y} \implies \neg A(x) \wedge \neg A(y)$$

However, just because x and y are not alone, it does not mean they are on the same side; this is only an implication.

As for the statements themselves:
"For Superman to be defeated, it has to be that he is facing an opponent [Batman] alone and his opponent is carrying Kryptonite."

$$D_s \rightarrow (A_s \wedge K_b)$$

"Acquiring Kryptonite, however, means that Batman [Superman's opponent] has to coordinate with Lex Luthor and acquire it from him."

$$K_b \rightarrow C_{l,b}$$

"If, however, Batman coordinates with Lex Luthor, this upsets Wonder Woman, who will intervene and fight on the side of Superman."

$$C_{l,b} \rightarrow U_w \wedge C_{w,s}$$

For $C_{w,s}$ to be true, it means the following will be true as well:

$$\neg A_s \wedge \neg A_w$$

b)
First, all three statements are combined with "AND".

$$[D_s \rightarrow (A_s \wedge K_b)] \wedge [K_b \rightarrow C_{l,b}] \wedge [C_{l,b} \rightarrow (U_w \wedge C_{w,s})]$$

Then, implication elimination is used on the three statements.

$$[\neg D_s \vee (A_s \wedge K_b)] \wedge [\neg K_b \vee C_{l,b}] \wedge [\neg C_{l,b} \vee (U_w \wedge C_{w,s})]$$