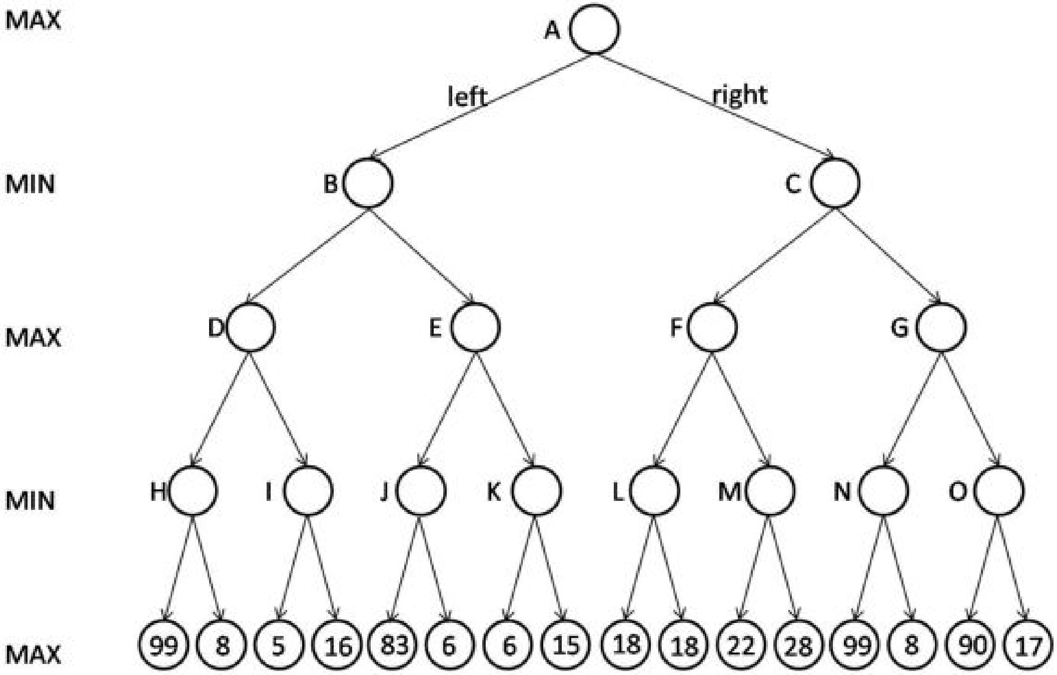
Netids: ajj64, rk605

Assignment 02

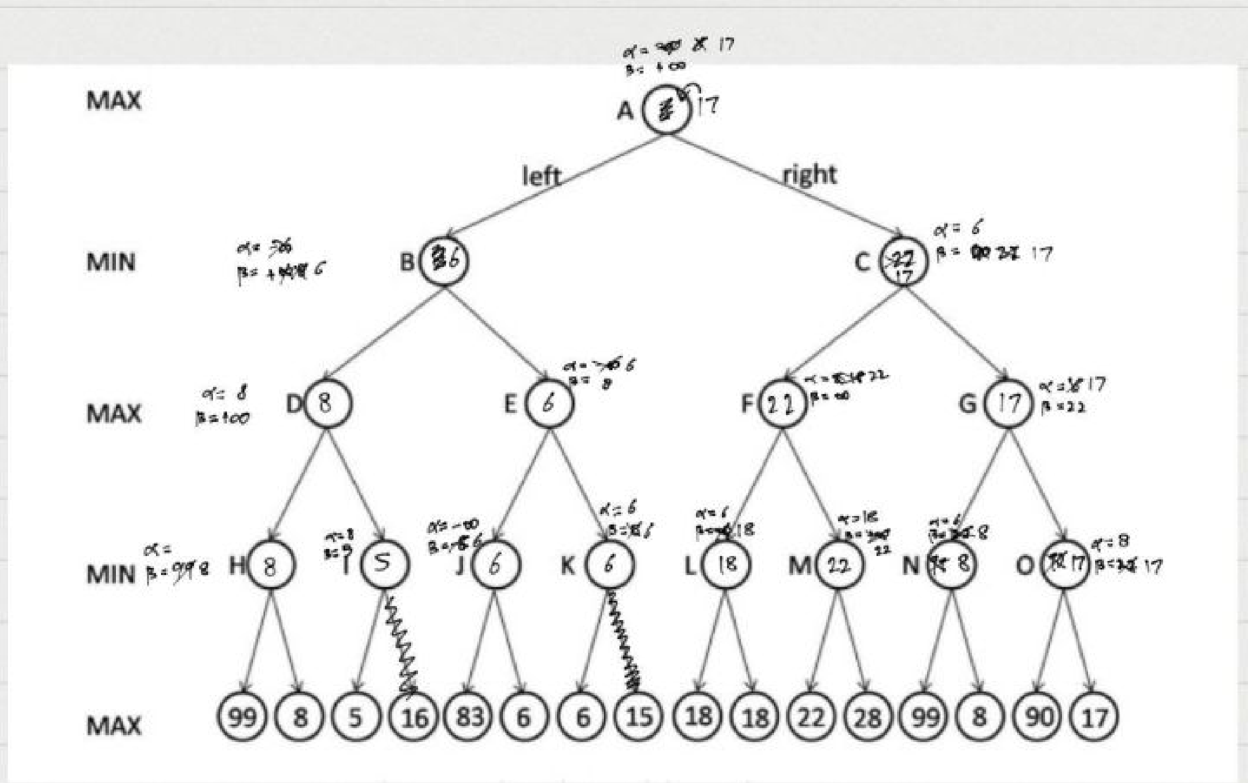
2.



a. The value of each intermediary node are as listed below:

H = 8, **I** = 5, **J** = 6, **K** = 6, **L** = 18, **M** = 22, **N** = 8, **O** = 17,
D = 8, **E** = 6, **F** = 18, **G** = 17,
B = 6, **C** = 17,
A = 17.

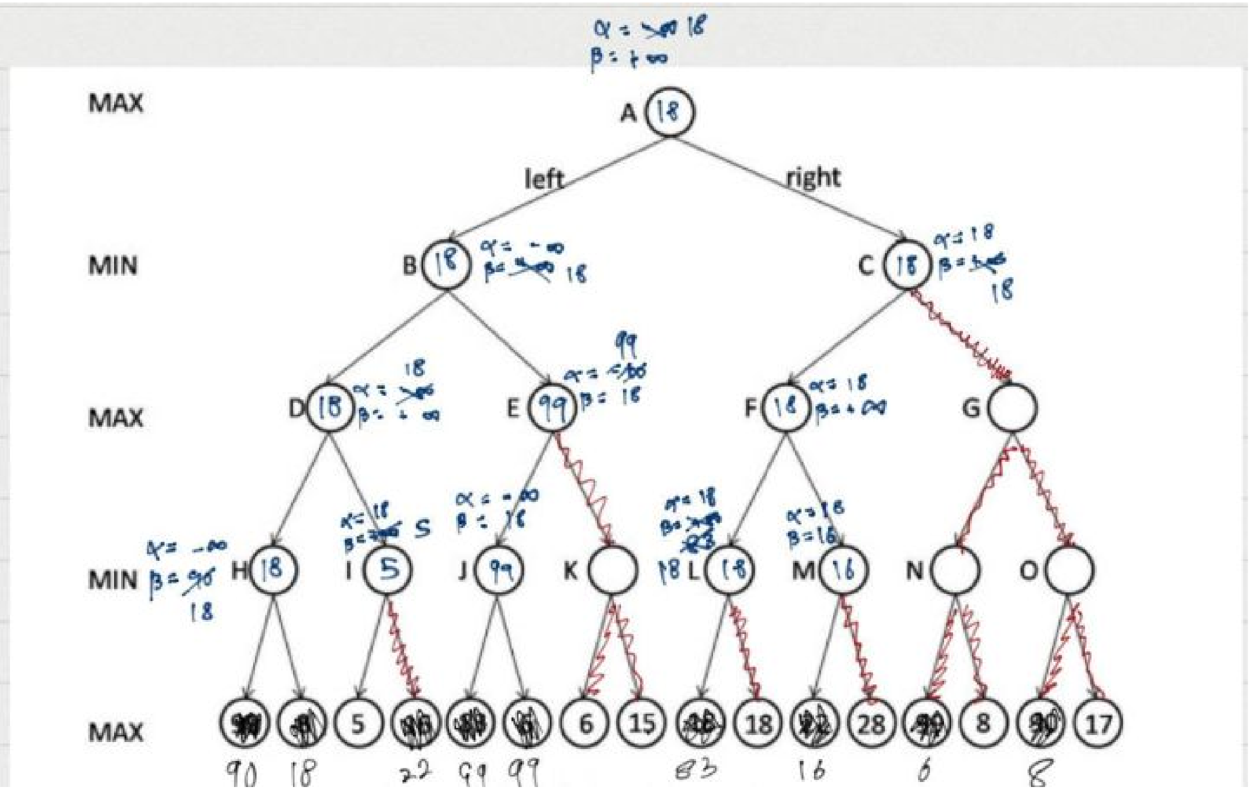
b. The screenshot below indicates that only the **16,** and **15** nodes are pruned (they are connected to nodes **I** and **K** respectively). The alpha / beta values are also listed, but for clarity, their final values will be listed below the picture.



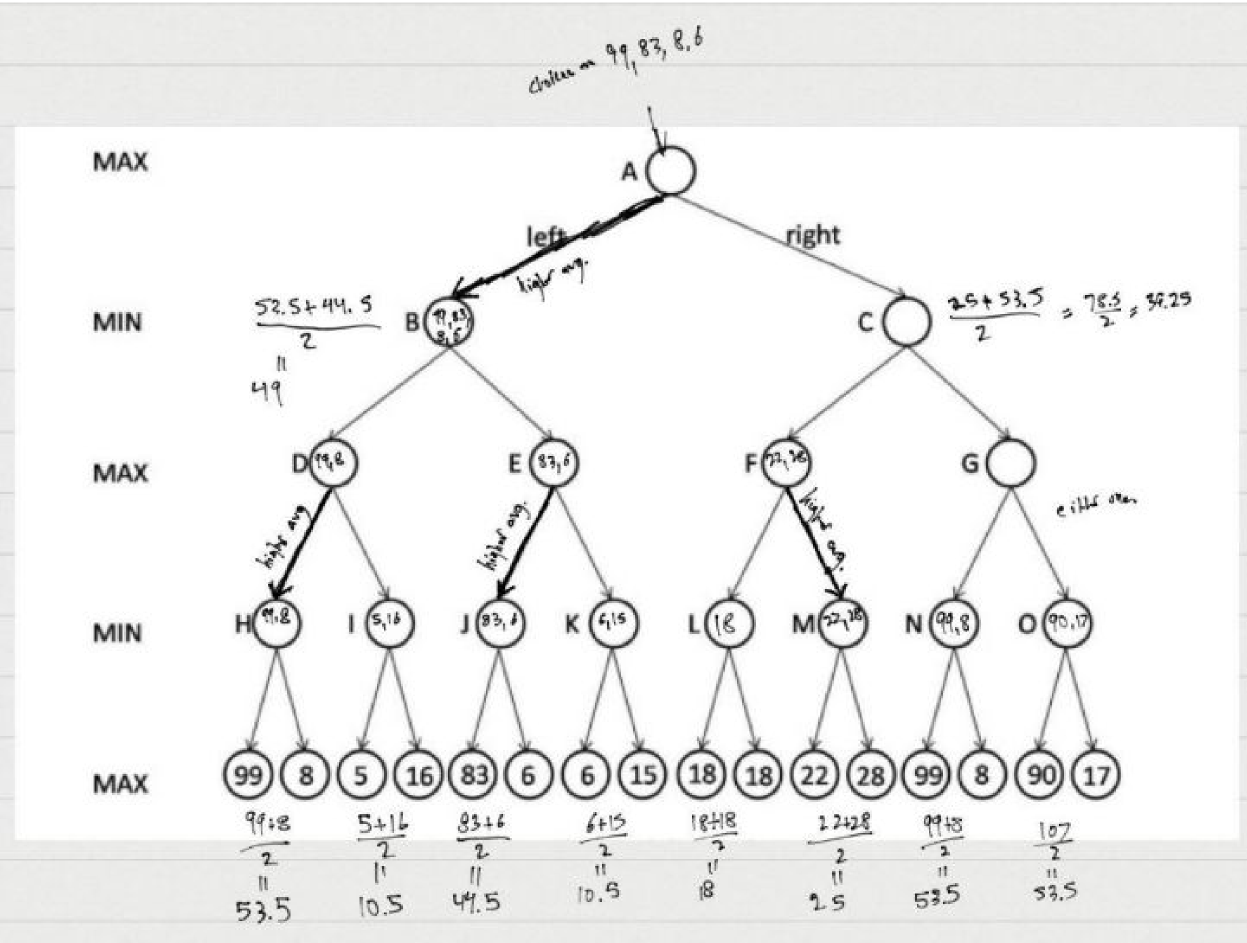
A:(α=17,β= +INF), **B**:(α= -INF,β=6), **C**:(α=6,β=17), **D**:(α= 8,β= +INF), **E**:(α=6,β=8), **F**:(α=22,β= +INF), **G**:(α=17,β=22), **H**:(α= -INF,β=8), **I**:(α=8,β=5), **J**:(α= -INF,β=6), **K**:(α=6,β=6), **L**:(α=6,β=18), **M**:(α=18,β=22), **N**:(α=6,β=8), **O**:(α=8,β=17)

c. The MAX player will choose 17 when employing the exhaustive minimax algorithm. The MAX player will also choose 17 when employing the minimax algorithm with alpha-beta pruning. In general, the only real difference between the exhaustive algorithm with that of alpha-beta pruning is that the latter attempts to save time by making the assumption that certain branches are not worth exploring. As a result, for the most part the two will have the same result, but while the exhaustive algorithm explores every available option to ensure the most optimal result based off its assumptions, the pruning method may exclude certain options that have the potential to change the final answer. This depends heavily on how the game tree's nodes are ordered.

d. The best optimization for alpha-beta pruning can be seen below:



e. Since this has now become a game of chance, I expect the MAX player to play as they have always (in the sense that they'll pick the highest utility value that they can). However, given that we cannot determine with absolute certainty what the MIN player will pick, we'll have to use the average values of the nodes as our choices for the MAX player to pick:



Thus, via the use of average utility values, MAX will still aim for the left side of the tree.

Ultimately, the choice of the MAX player depends on what the MIN player ends up picking each time. Since we know the MIN player has a 50 / 50 chance of picking either a left or right child node, we can then approximate the probability of the MAX player choosing a terminal node, these being: 25% for 99, 25% for 8, 25% for 83, and 25% for 6.

In terms of alpha-beta pruning, this scenario would not be plausible, as we cannot actively

3.

a. The set of variables are basically each of the 81 individual cells in the 9x9 sudoku grid (you can interpret this to mean that each state contains a single variable N, or each state is in itself a unique variable (meaning that the cell at row 1, col 1 is different from that of row 2, col 3)).

The domain of each cell is the set of all positive integers between 1 and 9 (that is to say that each cell contains exactly one positive integer, and that number must be within the given domain).

The constraints are basically this:

1. Every column and row must contain digits from 1 to 9.
2. Each of the nine 3x3 boxes in the puzzle must contain every number from 1 to 9.
3. Within each 3x3 box, every filled cell must be distinct with one another (i.e the box is filled with a single 1, 2, 3, 4, 5, 6, 7, 8, 9, meaning no repeats).
4. A number in a particular row and/or column must be unique to that specific row and/or column (i.e if there's a 9 in row 1, column 1, there cannot exist another 9 anywhere else in either row 1 or column 1).

b. Start state: any of the empty, unfilled cells in the 9x9 grid.

Successor Function: given a cell $N_{i,\,j}$ (where *i*, *j* represent the row, column), this function returns a set of (#value, #next_available_state) ordered pairs, where each #value is taken from the domain of $N_{i,\,j}$ (which is any integer between 1 and 9), and the #next_available_state being the next unfilled cell the agent shall encounter.

Goal Test: Check if a given state (cell, in this case) is valid based off given constraints (those being that every row/column/3x3 box contains a distinct value between 1 and 9). If this is the case, then continue on to the next state, and repeat the process until every state has been verified.

Path Cost: given a state (cell, in this scenario), we return the numerical representation of number of potential values *lost* (i.e made no longer valid given the current state's intended value) for every other unfilled cell in the 9x9 grid. This is the only real 'cost' that I can see being plausible for a problem such as sudoku.

In regards to the backtracking search algorithm, the *minimum remaining values heuristic* would be most effective, because of the other heuristic (maximum degree) does not really contribute to determining a state's given value, as the number of constraints is virtually the same for every unfilled cell in the 9x9 grid. The remaining values heuristic increases in utility as the number of prefilled cells (M) increases; however, the degree heuristic does not. This is due to the former potentially decreasing the branching factor for each cell, leading to a quicker solution than the latter could.

The branching factor in this case is 9 (it's the maximum number of available values for each unfilled cell).

The solution depth would be equivalent to the number of unfilled states in the 9x9 grid (it's 81 if the grid is completely empty, and 81-M if we have an M number of pre-filled values).

The maximum depth of the search space is pretty much our solution depth, but in the worst case scenario. That would be 81 (meaning that if each node represented a given cell's potential value, the path to the solution from that initial state would be 80 nodes deep (81 if you counted the initial) to account for filling in every other cell, and its potential value.

In terms of the state space, there are a total of 81 cells in a 9x9 grid, with each cell having a potential of 9 different values. Therefore, the number of states that exist is at most 9^81, or 9^(81-M) if the number of prefilled numbers M is >0.

c. The difference between a “hard” and “easy” sudoku problem is the heuristic value M, which is the numbers that have already been filled in the 9x9 grid. The reason here, is that (based off the initial introduction to Sudoku that this problem gave us) there is a single valid solution for every game. Therefore, given a higher heuristic value for M, we can determine the values of the remaining unfilled cells more easily by basing our potential values off the pre-filled ones (i.e making it “easier” to find a solution, as unfilled cells will have fewer values to choose from).

d. A local search algorithm can be as follows:

```
Loop (int variable named row from 1 to 9)
{
        Loop (int variable named col from 1 to 9)
        {
                Check if cell at row, col is filled or not
                        If unfilled, assign random integer value from 1 to 9.
                        Else, ignore.
        }
}
checkForErrors() {
//iterate through all rows, checking for errors/constraint breakers
        //if error is found, break and return row & col of invalid cell.
//iterate through all columns, checking for errors/constraint breakers
        //if error is found, break and return row & col of invalid cell.
//iterate through all nine 3x3 blocks, checking for errors/constraint rebels
        //if error is found, break and return row & col of invalid cell.

//return (99,99) (this is only reached if previous checks do not turn up positive)
}
while (checkForErrors does not return (99,99) )
{
//Change value of invalid cell (whose coordinates are found via checkForErrors()
to a random integer from 1 to 9
}
```

The aforementioned semi-pseudo code would ideally work less effectively on “easy” (M value/ number of pre-filled cells is higher) problems than an incremental search algorithm would, because the former would still assign values to unfilled cells randomly / hopefully try to get solution, while the latter would be able to find a solution more intelligently. Consequently, for “hard” problems (M value is much lower), I feel that my algorithm would be more effective at finding a valid solution, even if by chance (random variable assignment), than the incremental search algorithm, because the heuristic value for the latter does not help it in a significant means to make it more efficient than the former.

4.

a. Dictionary:

s: superman
b: batman
l: lex luthor
w: wonder woman
K(x): x has kryptonite
D(x, y): x defeats y
W(x, y): x works with y

Premises:

1. (~W(w, s) & K(b)) -> D(b, s)
2. W(b, l) -> K(b)
3. W(b, l) -> W(w, s)

Knowledge Base:

((~W(w, s) & K(b)) -> D(b, s)) & (W(b, l) -> (K(b) & W(w, s)))

b. Convert the knowledge base from part A to 3-CNF form:

(~(~W(w, s) & K(b)) V D(b, s)) & (~W(b, l) V (K(b) & W(w, s)))

((W(w, s) V ~K(b)) V D(b, s)) & ((~W(b, l) V K(b)) & (~W(b, l) V W(w, s)))

This gives us the following formula:

(W(w, s) V ~K(b) V D(b, s)) & (~W(b, l) V K(b)) & (~W(b, l) V W(w, s))

c. Prove that Batman cannot defeat Superman through resolution inference:

P1. ~K(b) V W(w, s) V D(b, s)
P2. ~W(b, l) V K(b)
P3. ~W(b, l) V W(w, s)
Prove: ~D(b, s)
Assume negated conclusion: D(b, s)
----

1. ~W(b, l) V W(w, s) V D(b, s)   P1, P2 Resolution
2. T.T #RIP.

5. In regards to the A* implementation, I would utilise a minimum spanning tree as a primary means of a heuristic. Ideally, h(n) =length of unvisited nodes in MST. This is because the visual concept of an MST is very similar to that of A* / dijkstra. The best example for this would be a road map, where the roads / distances between each city are laid out. To implement this function, I would have to calculate and store the values of all the *not yet visited* cities together (to determine how much ‘mileage’ I have left to optimize a path out of). I consider this to be an

effective heuristic in that we can map out the route that utilises the least amount of ‘mileage’ (because this means that it is the shortest valid path).

a. For my algorithm (which implemented the Euclidian distance as a cost and an MST as the heuristic):
For 10 cities, all 25 files ran within the time limit.
For 25 cities, all 25 files ran within the time limit.
For 50 cities, I got 3-7 files running within the time limit (regenerated files and ran program again 3 times)
For 100 cities, 1 file ran within the time limit (regenerated files and reran program 3 times)

b. The average runtime for all 25 files of 10 cities was: 2.272ms.
The average runtime of all 25 files of 25 cities was: 13683.132042ms
The average runtime of all 25 files of 50 cities was: 419690.7954ms
The average runtime of all 25 files of 100 cities was: 710567.4355ms (it easily exceeded the 10 minute limit of the others, so I think I got lucky with one).

c. All nodes were generated for all 25 files of 10 cities (within the 10 minutes)
All nodes were generated for all 25 files of 25 cities. (within the time limit)
Around forty percent of the nodes were generated throughout the 25 files of 50 cities (within the 10 minutes)
Around eighteen percent of total nodes were generated throughout the 25 files of 100 cities within the time limit

d.

In general, A* showed me promising results, but exponentially took more time as our state space increased (i.e by the time we reached 50 cities, the number of files running successfully dropped significantly from the initial 10-city challenge). I wrote my program in Java, so that may have contributed to my seemingly excessive runtimes.

6.

a. If we define h3(n) = min(h1(n), h2(n)), then we know that h3(n) is admissible if either h1(n) or h2(n) are admissible, as if h1(n) <= g(n), then min(h1(n), infinity) <= g(n). Similarly, if one of h1(n) or h2(n) is consistent, then h3(n) is consistent, as h3(n) = min(h1(n), infinity) <= min(c(n, n’) + h1(n’), infinity), which means that h3(n) is similarly consistent as h1(n).

b. If h3(n) = max(h1(n), h2(n)), then h3(n) is admissible if both h1(n) and h2(n) are admissible, as if both h1(n) and h2(n) <= g(n), then max(h1(n), h2(n)) <= g(n). In the same manner, h3(n) is consistent if both h1(n) and h2(n) are consistent. This can be shown as h3(n) = max(h1(n), h2(n)) <= max(c(n, n’)+h1(n’), c(n, n’)+h2(n’)) <= max(h1(n’), h2(n’)) + c(n, n’) = h3(n’) + c(n, n’), which makes h3(n) consistent.

c. If h3(n) = w*h1(n) + (1-w)*h2(n), where 0 <= w <= 1, then there are four separate cases. If only h1(n) is admissible, then h3(n) is admissible only when w=1, as this removes the h2(n) term. If only h2(n) is admissible, then h3(n) is admissible only when w=1, as this removes the h1(n) term. If both are admissible, then h3(n) is also admissible for all values of w. Finally, if neither h1(n) or h2(n) are admissible, then h3(n) is also not admissible. Like the problems above, the requirements for consistency follow the same pattern.

d. This algorithm is guaranteed to be optimal for values of w where 0 <= w <= 1. This is because when g(n) is inflated, there is no effect, but w must be lower than when it is multiplied with h(n), as if w is higher than 1, it would cause the heuristic to overestimate, and thus makes it not admissible. When w=0, the algorithm becomes f(n) = 2*g(n), which is essentially a uniformed best first search. When w=1, the algo becomes f(n) = g(n)+h(n), which is simply the classical A* search. When w=2, the algo is f(n) = 2*h(n), which is then a greedy best-first search.