Question 1

a) Local Search Operators

From a given tour, it selects 2 random cities on the tour (other than the starting and ending city) and reverses the path in between those 2 cities

For example, for a 10-cities problem if the current state is:

A->C->B->E->F->I->J->D->H->G->A

The operation selects 2 random cities on the tour, say E and D,

A->C->B->**E**->F->I->J->**D**->H->G->A

then the successor is:

A->C->B->**D->J->I->F->E**->H->G->A

The first and last city, which is always A, is never selected to preserve the tour

The local search operator that is used can be seen in the method TSPState.getSuccessor() method in tsp_state.py

b) Annealing Schedules

Experimented with 3 different Annealing Schedules:

1. **Exponential Schedule**[1]

$$T(t) = \alpha^t T_0 \quad \text{where } 0 < \alpha < 1$$

For 36-city problem, chose $T_0 = 200000$ and $\alpha = 0.999995$

2. **Logarithmic Schedule**[2]

$$T(t) = T_0 / (1 + \alpha \text{Log}(1 + t)) \quad \text{where } \alpha > 1$$

For 36-city problem, chose $T_0 = 10000$ and $\alpha = 1000$

3. Quadratic Schedule[3]

$$T(t) = T_0 * ((maxSteps - t)/maxSteps)^4$$

Where maxSteps is the maximum number of steps the algorithm will be run
For 36-city problem, chose $T_0 = 2000000$ and $maxSteps = 2000000$

> Note: The Temperature vs Time plot for all three schedules can be seen in 'temperature-plots' directory. Please take a look

Findings

- **Motivation for choosing the 3 schedules**: Tried out various other annealing schedules(other than the 3 mentioned here) like Linear ($T(t) = T_0 - \alpha t$) and found that the best results could be achieved with a curve that:
    - starts at high initial temperature with high cooling rate, and then
    - cooling rate decreases towards the middle. But the temperature is decreased slowly
    - Stays at very low temperature for a long time, due to very small cooling rate
- Setting the initial temperature high allows the search to explore more (exploration vs exploitation) and this helps avoid local minima. At this point bad solutions are accepted more and more. Once the temperature approached towards zero, almost all bad solutions are rejected and the cost keeps on dropping (exploitation)
- At intermediate temperatures, bad solutions are accepted and rejected with almost equal probability
- The performance of the search depends on the schedule used. Only when the schedule lowers the temperature slowly and stays at lower temperatures for a longer time, do we get a good enough solution close to the optimal solution
- From the cost vs time plots of all the schedules, a substantial decrease in the cost only started occurring once the temperatures approached towards 0

Note: The Cost vs Time plot for all three schedules can be seen in 'cost-plots' directory. Please take a look

- The parameters of the schedule, such as T_0, α and maxSteps had to be fine tuned for the 36-city problem to bring it closer to the best solution. I ran the 36-city problem on various online TSP solvers and the best result I got was 464 km. All the 3 schedules reached very close to this solution

Note: The result for each schedule can be seen in 'results' directory. Please take a look

- An interesting observation is that the Logarithmic Schedule was able to achieve the same result faster than the other two schedules. As can be seen from the temperature plots, the Logarithmic Schedule is steeper at the beginning and then flattens for a long time

What Schedule would I choose?

I got the best results in the lowest time with the Quadratic Schedule. So ideally I would choose a schedule which:

- Starts out at high temperature which would allow me to explore various options and escape local minima
- That decreases the temperature slowly
- and stays in the lower temperatures for a long time which increases my chances of closing in on the optimal solution

c)Running on 36-cities problem

Note: The algorithm was run on an Acer laptop with i7 processor and windows platform. Running this on a different system can lead to different results.
To run the program, edit anneal.py and set *inputFile* variable to the path of the problem file, and set *selectedSchedule* variable to 'exponential', 'logarithmic' or 'quadratic'. Save and execute the python file

For results of each schedule, please see 'cost-plots' and 'results' directory.

To summarize, the results I got for the three schedules (terminating after 5 minutes):

1. Exponential Schedule
    a. Cost = 467.50
    b. Tour = A->AB->N->V->X->D->L->Y->AH->AI->I->T->AF->AA->G->B->AJ->H->AG->M->R->AE->F->S->U->AC->C->E->AD->O->P->Z->Q->J->W->K->A
2. Logarithmic Schedule
    a. Cost = 465.84
    b. Tour = A->AB->N->V->X->D->L->AH->AI->I->Y->T->AF->AA->G->B->AJ->H->AG->M->R->AE->F->S->U->AC->C->E->AD->O->P->Z->Q->J->W->K->A

3. Quadratic Schedule
    a. Cost = 465.76
    b. Tour = A->K->W->J->Q->Z->P->O->AD->E->C->AC->U->S->F->AE->R->M->AG->H->AJ->B->G->AA->AF->T->Y->I->AI->AH->L->D->V->X->N->AB->A

d) Complete?

Yes. As the algorithm will not get trapped in local minima and will converge towards the optimal solution, if the temperature is decreased slowly

e) Optimal?

Yes.
Theoretically, If the schedule lowers T slowly enough, the algorithm will find a global optimum with probability approaching 1 (Russel & Norvig)

For all of the test runs, the algorithm was terminated after 5 minutes and the best cost obtained is 465.76 for the tour:
A->K->W->J->Q->Z->P->O->AD->E->C->AC->U->S->F->AE->R->M->AG->H->AJ->B->G->AA->AF->T->Y->I->AI->AH->L->D->V->X->N->AB->A

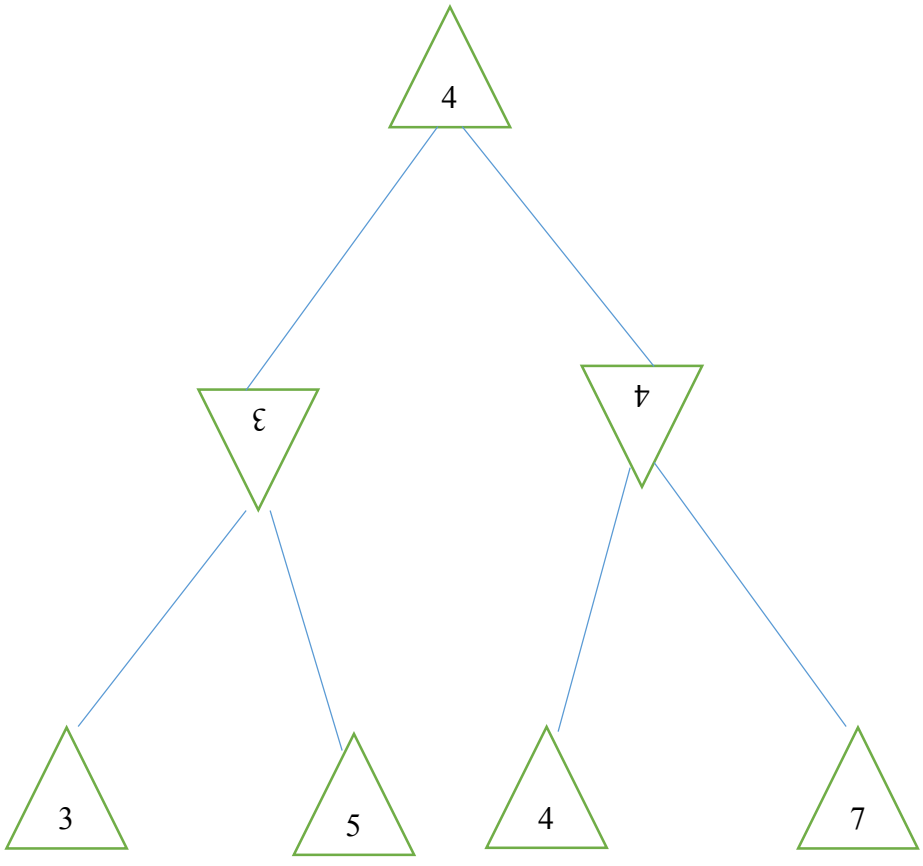
When verified using online TSP solver[4] the solution obtained was the same as the solution above which I got

Question 2

a)

If MIN plays sub-optimally then that means MIN selects a move with minimax utility greater than or equal to the move predicted by minimax. Since MAX takes the maximum over these values, then the minimax utility against a suboptimal MIN is
greater than or equal to the minimax utility against an optimal min.

For example, for the below game tree the minimax value at root is 4 when MIN plays optimally



If suppose plays sub-optimally and selects 5 instead of 3 at the leftmost MIN node, then the minimax value at root becomes 5 (which is greater than the value against optimal MIN) as shown by below game tree

b)

The game tree above shows the values if MAX and MIN both played optimally. As shown in the tree, the optimal move for MAX is to select the move that leads to the leaf with utility 3. But if MAX knows that MIN will play sub-optimally, then MAX knows that MIN can select 9 at the rightmost MIN node. This allows MAX to make a sub-optimal move and select the other option, which would get MAX a utility value of 2 against an optimal MIN, but will get 9 against a suboptimal MIN

Question 3

a)

Minimax will work as usual if it's set up right. Each terminal node will now have a vector of values [U_{MAX}, U_{MIN}] where U_{MAX} is the utility of that node for the MAX player and U_{MIN} is the utility of that node for the MIN player. We will be backing up a vector of evaluations and at each level the player will choose what is best for him, even if it is also good for the other player. Thus if we assign increasingly positive values for states increasingly better for MAX and increasingly negative values for states increasingly better for MIN, then minimax will work unmodified. If both players have increasingly positive values, each player just picks the maximum value, so it becomes a "maximax" algorithm

However, alpha-beta pruning will not work because in alpha-beta pruning we assume that what's good for max is bad for min – for example min won't let max go down a path since min can force something worse, so max knowing this doesn't have to explore that path. But without zero-sum assumption, the same state could be good for both min and max; you can't assume that just because max likes it that min won't, and vice versa.

b)

Alpha-Beta pruning cannot be done in this case. If there no constraints on the two utility functions, there their values can be anything and there would be no relation between U_A and U_B for any node. So we cannot decide if MIN going down a path would lead to a better or worse alpha, or MAX going down a path would lead to a better or worse beta. And any terminal node that is not yet visited could be optimal for both players

c)

Yes. Alpha-Beta pruning can be done in this case. For any terminal node, if U_A is the utility for player A (say MAX) then the utility for player B (MIN) is given by the range

$$U_B = [U_A - k, U_A + k]$$

Which means when MAX maximizes U_A, U_B also gets maximized. Also given U_B we can know the upper and lower bounds of U_A and vice versa. This allows us to know the possible alpha and beta values that can be obtained by going down a path in the game tree

So for a zero-sum game, we would do pruning as below:
At a MIN node, if value at that node <= alpha then do pruning
At a MAX node, if value at that node >= beta then do pruning

For this particular case, we could do pruning as below:
At a MIN node, if value at that node – k <= alpha then do pruning
At a MAX node, if value at that node + k >= beta then do pruning

Question 4

c) Better evaluation function

The evaluation function used takes into account 3 factors:

1. The strategic winner for a given board, which is determined based on calculation of ODD and EVEN threats[5][6] for each player [see get_strategic_winner() method]
2. Winning early will evaluate more than winning late. Also losing late will evaluate more than losing early [from focused_evaluate]
    a. Whether a game is won/lose early/late is determined by the number of remaining empty slots in the board
3. Prefer your pieces in the center than the edges. Prefer opponent's pieces towards the edges than in the center [from basic_evaluate(). But I have increased the points for this factor by multiplying by 50]

A **threat** is three pieces in a chain (horizontal, vertical or diagonal) with the fourth square open. A threat is an odd threat if its on an odd row and it is an even threat if it is on an even row. A threat is shared (or mixed) if it is a threat for both players.

Analysis of the odd and even threats leads to some interesting conclusions which are used in the better_evaluate() function:

- A player with more number of odd threats is likely to win.
- An even threat is only useful for P2, and it can win with even threats only under some conditions
    - If both players have same no of odd threats, then P2 can win only
        - if there are some shared odd threats OR
        - If there is no shared odd threat, but there are some number of even threats for P2. If there are no even threats for P2, then it is a draw
    - If P2 has an extra odd threat than P1, then
        - P2 can win if
            - If there are odd number of shared threats OR
            - No shared threats, but some number of even threats. If no even threats for P2 then it is a draw
        - P1 can win if there are even number of shared threats

References

1. http://iopscience.iop.org/article/10.1088/0305-4470/31/41/011/pdf
2. Aarts, E.H.L. & Korst, J., 1989
3. Modified from http://iopscience.iop.org/article/10.1088/0305-4470/31/41/011/pdf
4. https://neos-server.org/neos/solvers/co:concorde/TSP.html
5. Victor Allis 1988 Master thesis http://www.informatik.uni-trier.de/~fernau/DSL0607/Masterthesis-Viergewinnt.pdf
6. https://en.wikibooks.org/wiki/Connect_Four