1. (12 points) Formulate the game of Sokoban as state space search (if you haven’t heard of it, check it up in your Linux implementation, or Google for it). Suitably define states, operators, cost functions, and the goal test. Ensure that your operators do not generate repeated states (readup AIMA section 3.5).

2. (8 points) Using the technique described in class for inventing heuristics, define an admissible heuristic for solving the above problem. Do not try to ‘be clever.’ Example: a heuristic such as $h(n) = 0$ will fetch you $h(n)$ points. Your solution should describe how the constraints on the original search problem have been relaxed to arrive at the heuristic.

3. (8 points) Problem 3.12 from your textbook. Readup on GRAPH-SEARCH as discussed in Fig. 3.19.

4. (10 points) Problem 4.3 (b) from your textbook.

5. (12 points) Problem 4.4 from your textbook. Suggest a ‘fix’ to return optimal solutions.

6. (15 points) It is well known that two heuristics for solving the TSP problem are: (i) the cheapest second degree graph going through the remaining nodes, and (ii) the minimum spanning tree (MST) through the remaining nodes. Explain how these heuristics can be derived by relaxations of the original problem statement. Give a careful analysis of the complexity of computing these heuristics.


8. (25 points) For this question, you are going to write a program for solving the 8-puzzle. But you should structure your code in such a way that it can play the 15-puzzle as well. Your program must implement A* with four heuristics: the three heuristics studied in class
   - $h_1 =$ number of misplaced tiles.
   - $h_2 =$ manhattan distance (i.e., sum of distances of tiles from their goal positions).
   - $h_3 =$ number of swaps with the empty board position (Gaschnig’s heuristic).
   and one more:
   - $h_4 =$ minimum number of column adjacent blank swaps to get all tiles in their destination column plus minimum number of row adjacent blank swaps to get all tiles in their destination row.
Observe that \( h_3 \) and \( h_4 \) are a bit more involved than \( h_1 \) and \( h_2 \) and require some careful algorithmic implementation.

To begin this question, notice that some 8- and 15-puzzle problems are not solvable (why?). So even if you run your perfect heuristic on a problem that is unsolvable, you will end up exhaustively searching the space before giving up. Therefore, create a collection of solvable test problems to pit against your program. This can be easily done by starting with some random tile layout, making a sequence of moves (randomly), and choosing the resulting layout as the goal state.

Then compare your implementations of A* search (with all the four heuristics) and also breadth first search (BFS; with \( h(n) = 0 \)). Present results of the following form (graphs or tables): (i) Number of nodes expanded by A*, for each heuristic, versus BFS, for both problem sizes, and (ii) Effective branching factor (see textbook; Section 4.2) for A*, with each heuristic, versus BFS, for both problem sizes.

To have confidence in your results, it is recommended that you average each measurement over multiple problems (of a given size).

Also explain if some of the heuristics dominate others. Give as tight a total or partial ordering as you can about the four heuristics.