CS561 AI Assignment:
Name:Dheeraj Khatri
Roll No:120101021

Exercise 1 & 2. Consider the 15-puzzle problem with Manhattan distance heuristic function.Apply A* and IDA* algorithms to the given problem with 20 randomly generated initial states and comment and compare their results. Tabulate the results as shown below and plot the same.

Apply IDA* and RBFS algorithms to the given problem with 20 randomly generated initial states and comment and compare their results. Tabulate the results as shown below

Initial State	Optimal Sol. length			Nodes generated		
	A*	IDA*	RBFS	A*	IDA*	RBFS
[[4 1 2 3] [5 13 6 7] [0 8 10 11] [9 12 14 15]]	11	11	11	56	30	60
[[8 5 2 3] [1 6 4 7] [9 13 10 11] [12 14 15 0]]	23	23	23	2807	2112	2308
[[4 9 1 2] [5 7 6 3] [8 14 13 10] [12 15 11 0]]	21	21	21	183	174	183
[[0 5 6 3] [13 2 4 7] [1 8 10 15] [9 12 11 14]]	27	27	27	1417	1335	1667
[[5 4 1 2] [6 9 10 3]	15	15	15	48	32	45

[8 13 11 7] [12 0 14 15]]						
[[8 2 3 7] [5 4 10 6] [1 13 9 11] [12 0 14 15]]	21	21	21	923	756	983
[[1 3 5 7] [4 2 10 6] [12 8 0 9] [13 14 15 11]]	19	19	19	75	54	87
[[0 2 7 6] [1 4 9 3] [8 10 5 11] [12 13 14 15]]	15	15	15	60	43	72
[[0 4 3 7] [5 2 1 6] [8 9 10 11] [12 13 14 15]]	11	11	11	25	23	25
[[0 1 2 3] [8 5 6 7] [12 4 10 11] [13 9 14 15]]	11	11	11	45	32	44
[[1 7 2 3] [5 0 6 11] [4 8 9 10] [12 13 14 15]]	15	15	15	204	136	248
[[8 7 0 11] [2 3 5 10] [4 1 14 15] [9 12 6 13]]	35	35	35	10624	7546	9641

[[1 12 0 3] [4 6 2 7] [9 5 10 11] [13 8 14 15]]	19	19	19	471	328	509
[[1 2 6 3] [9 8 4 7] [12 14 0 5] [10 13 15 11]]	23	23	23	224	170	267
[[6 4 3 7] [1 9 2 5] [8 13 10 11] [12 14 15 0]]	19	19	19	118	98	140
[[4 1 2 3] [8 7 11 0] [12 5 10 6] [13 9 14 15]]	15	15	15	198	173	223
[[2 9 6 3] [4 1 5 7] [8 10 0 11] [12 13 14 15]]	15	15	15	224	154	220
[[4 1 3 11] [5 2 9 6] [8 7 0 15] [12 14 13 10]]	25	25	25	2509	2385	2849
[[1 8 2 3] [12 4 6 7] [9 10 14 11] [5 13 15 0]]	21	21	21	582	479	638
[[2 4 6 7]	25	25	25	296	194	296

[1 13 3 9] [8 14 5 11] [12 10 15 0]]						

A* Algorithm:

```
function A*(start, goal)
    // The set of nodes already evaluated.
    closedSet := {}
    // The set of currently discovered nodes still to be evaluated.
    // Initially, only the start node is known.
    openSet := {start}
    // For each node, which node it can most efficiently be reached from.
    // If a node can be reached from many nodes, cameFrom will eventually contain the
    // most efficient previous step.
    cameFrom := the empty map

    // For each node, the cost of getting from the start node to that node.
    gScore := map with default value of Infinity
    // The cost of going from start to start is zero.
    gScore[start] := 0
    // For each node, the total cost of getting from the start node to the goal
    // by passing by that node. That value is partly known, partly heuristic.
    fScore := map with default value of Infinity
    // For the first node, that value is completely heuristic.
    fScore[start] := heuristic_cost_estimate(start, goal)

    while openSet is not empty
        current := the node in openSet having the lowest fScore[] value
        if current = goal
            return reconstruct_path(cameFrom, goal)

        openSet.Remove(current)
        closedSet.Add(current)
        for each neighbor of current
            if neighbor in closedSet
                continue        // Ignore the neighbor which is already evaluated.
            // The distance from start to a neighbor
            tentative_gScore := gScore[current] + dist_between(current, neighbor)
            if neighbor not in openSet  // Discover a new node
                openSet.Add(neighbor)
            else if tentative_gScore >= gScore[neighbor]
                continue        // This is not a better path.

            // This path is the best until now. Record it!
            cameFrom[neighbor] := current
            gScore[neighbor] := tentative_gScore
            fScore[neighbor] := gScore[neighbor] + heuristic_cost_estimate(neighbor, goal)

    return failure

function reconstruct_path(cameFrom, current)
    total_path := [current]
    while current in cameFrom.Keys:
        current := cameFrom[current]
        total_path.append(current)
    return total_path
```

IDA* Algorithm:

```
node              current node
g                 the cost to reach current node
f                 estimated cost of the cheapest path (root..node..goal)
h(node)           estimated cost of the cheapest path (node..goal)
cost(node, succ)  step cost function
is_goal(node)     goal test
successors(node)  node expanding function

procedure ida_star(root)
  bound := h(root)
  loop
    t := search(root, 0, bound)
    if t = FOUND then return bound
    if t = ∞ then return NOT_FOUND
    bound := t
  end loop
end procedure

function search(node, g, bound)
  f := g + h(node)
  if f > bound then return f
  if is_goal(node) then return FOUND
  min := ∞
  for succ in successors(node) do
    t := search(succ, g + cost(node, succ), bound)
    if t = FOUND then return FOUND
    if t < min then min := t
  end for
  return min
end function
```

RBFS Algorithm:

function RECURSIVE-BEST-FIRST-SEARCH(*problem*) **returns** a solution, or failure
  return RBFS(*problem*, MAKE-NODE(*problem*.INITIAL-STATE), ∞)

function RBFS(*problem*, *node*, *f_limit*) **returns** a solution, or failure and a new *f*-cost limit
  if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
  successors ← [ ]
  for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
    add CHILD-NODE(*problem*, *node*, *action*) into *successors*
  if *successors* is empty **then return** *failure*, ∞
  for each *s* **in** *successors* **do** /* update *f* with value from previous search, if any */
    s.*f* ← max(*s*.*g* + *s*.*h*, *node*.*f*))
  loop do
    best ← the lowest *f*-value node in *successors*
    if *best*.*f* > *f_limit* **then return** *failure*, *best*.*f*
    alternative ← the second-lowest *f*-value among *successors*
    result, *best*.*f* ← RBFS(*problem*, *best*, min(*f_limit*, *alternative*))
    if *result* ≠ *failure* **then return** *result*

Comment: Optional solution length for all 3 algorithm is found same.
If use any other non consistent heuristic then the number of generated nodes increases.
Bonus Problem: (Weighted heuristic)
H(n) = w*H(n)

H(n):Hamming distance of current state n and goal state
W: if we randomly select some position out of 16, if 0(blank) tile is present there then its 1

Exercise 3:Generate at least 20 instances of 8-queens and solve them using random restart hill-climbing and simulated annealing. Measure the percentage of solved problems and plot. Comment on your results.

Note: In 8 Queen Problem I have considered the version , which was discussed in the class, where each column contains one queen at the starting state. Initial states are generated randomly.

ALGO: Random Restart

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

current ← MAKE-NODE(*problem*.INITIAL-STATE)
loop do
neighbor ← a highest-valued successor of *current*
if neighbor.VALUE ≤ current.VALUE **then return** *current*.STATE
current ← *neighbor*

Figure 4.2 The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h.

Random restart conducts a series of hill climbing searches until goal is found. If each hill climbing has success probability p then expected number of restarts required is 1/p.
In random restart we are percentage of solved problems depend on how many restart we are considering. Here is some analysis with different restarts:

Restarts	Instances taken = 20

1	Solved = 1 (10%)
10	Solved = 17 (85%)

100	Solved = 100(100%)

Comment: We observe that as we increase the number of restarts of hill climbing percentage of solved instances increase, which is desired. As mentioned in the book p~0.14, number of restarts required are 1/p~7, we get the similar results with 85% success rate with 10 restarts.So we can say that radnom restart is very effective approach for 8queen problem, if number of queens are very large even then random restart hill climbing can solved very efficiently.

ALGO: Simulated Annealing

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(problem.INITIAL-STATE)
  for t = 1 to ∞ do
      T ← schedule(t)
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← next.VALUE – current.VALUE
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(ΔE/T)
```

Figure 4.5 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. Downhill moves are accepted readily early in the annealing schedule and then less often as time goes on. The *schedule* input determines the value of the temperature T as a function of time.

Temperature Schedule:
newTemprature = alpha * oldTemprature
T = 1
T_min = 0.00001
Where is alpha is taken as 0.9

ALPHA	0.9	0.5	0.1
Solved Instances:	19(95%)	16(80%)	12(60%)

Comment: As mentioned in the book if we temprature schedule decreases the temperature slowly enough than algorithm will find the optimal solution with probability approaching 1.

Exercise 4. Solve 8-queens problem using genetic algorithm with single point cross over, roulette wheel selection and very small mutation probability. Comment and compare your results for various instances of initial population.

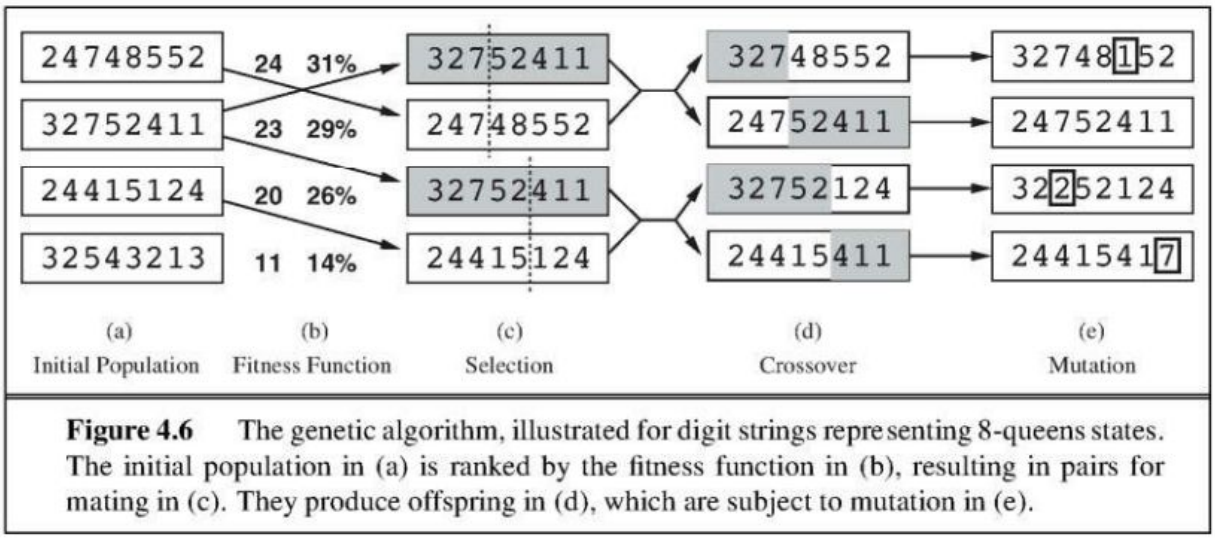

Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

ALGO: Genetics algorithm start with some initial population which is randomly generated then uses roulette wheel selection to choose the population which will go in next generation. After selection one point crossover is applied for which point is chosen randomly after that with very less probability mutation is applied to cross overed generation. This process continues until we find the solution of given time is elapsed (ie: No solution is found in the given time).

Fitness function is taken as how many queens are in non attacking position, higher values of fitness represents better state. Probability of being chosen for reproducing is directly proportional to the fitness score.

Mutation Probability = 0.1
Initial population = 16
Max time allowed = 30000
Solved instances = 17

Time taken for 20 instances:

14309	6772	2430	2869	28361	None	6345	6165	19396	11915
4845	9991	9729	19520	24818	5257	3964	None	None	8565

None: represents instance is not solved in the given time.
Comment: If we increase the time the at some time above algorithm will definitely solved the problem. If we increase the initial population then time required decreases. If average fitness score of population increases then we are capturing good population with time.

Exercise 5. Construct a game playing agent using alpha-beta pruning for tic-tac-toe game. Assume your own move generators and evaluation function. What is the effective branching factor? Can you improve this by improving the move ordering? Comment on your results.

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a), α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

Effective Branching factor with alpha beta pruning: Branching factor can be calculated as $N^{(1/d)}$ Where:
N : Number of nodes generated at the starting = 30690
D: max depth = 9
Effective Branching factor: 3.15

When we apply move ordering number of generated nodes decreases.We get follwing results for move ordering:
N = 28620
D = 9
Effective Branching factor:3.12

Comment: Move ordering selection basicly choses successor randomly so it decrease the nodes generated and we can see that code follows the same thing and leads to reduced branching factor.