CS410: Artificial Intelligence 2021 Fall
Homework 1: Search Algorithms
Due date: 23:59:59 (GMT +08:00), October 10 2021

1. Consider the 8-queens problem. Your goal is to place 8 queens in a chessboard so that no two queens are in the same row, column, or diagonal. Recall that to formulate it as a search problem, we need to specify several components.

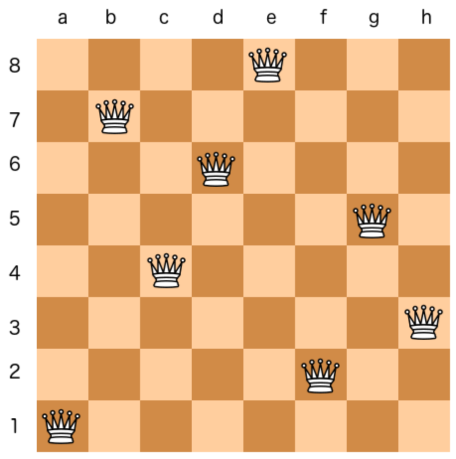


Figure 1: A feasible solution for the 8-queens problem.

   (a) Support we formulate the state space as the set of all arrangements of $0, 1, 2, \ldots, 8$ queens on the board. What would each component of the search problem be? How many states are there?

   (b) Give an alternative formulation which has a much smaller state space. List all the components of the search problem, and specify the number of states.

Solution:

(a) i. **States**: A number $0 \leq n \leq 8$ and n queens arranged on the board.

   ii. **Actions**: Place a new queen on a certain position on the board which the former queens are not on.

iii. **Successors**: Place the $(n+1)^{th}$ queen on an empty position and update the arrangements if $n < 8$.

iv. **Goal test**: There are 8 queens on the board and none of them are in the same row,column or diagonal.

v. **State number**: The number of states with n queens on the board is choose n positions on the board. So the total number of states S is:

$$S = \sum_{i=0}^{8} \frac{64!}{(64-i)!} \tag{1}$$

(b) Here we place the queens on each row one by one. For instance, we place the n^{th} queen on the n^{th} row at the n^{th} times. At the same time, we avoid to place the queen on the same column to the queen placed before.

i. **States**: A number $0 \leq n \leq 8$ and a sequence of n numbers $s = s_1 s_2 \ldots s_n$ means place n queens on the first n columns.

ii. **Actions**: Place a new queen on the $(n+1)^{th}$ column.

iii. **Successors**: Append a new number s_{n+1} at the end of s means placing the $(n+1)^{th}$ queen at the $n+1$ column and s_{n+1} row. And s_{n+1} satisfies:

$$\forall 0 \leq i \leq n, s_{n+1} \neq s_i \tag{2}$$

iv. **Goal test**: $n = 8$ and the sequence s satisfies:

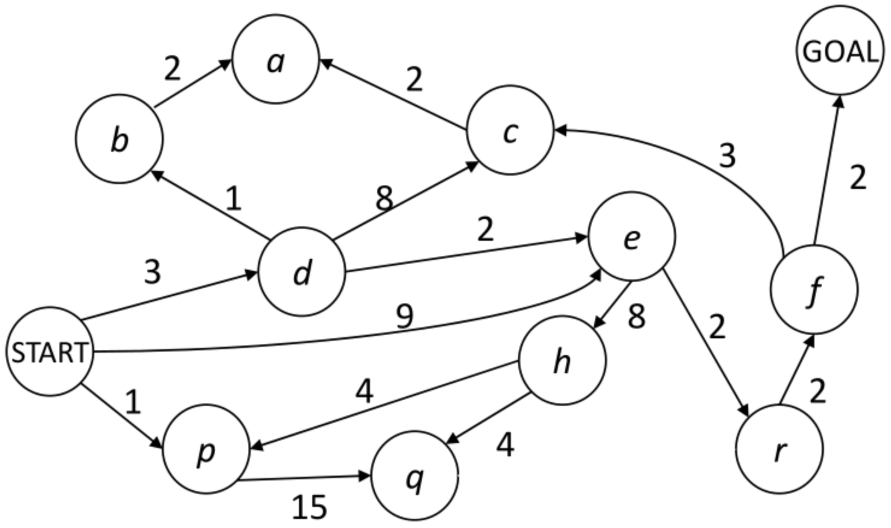
$$\forall 0 \leq i < j \leq 8, j - i \neq |s_j - s_i| \tag{3}$$

v. **State number**: The number of states with n queens on the board the number of the injections that from $\{1, 2 \ldots n\}$ to $\{1, 2 \ldots 8\}$. So the total number of states S is:

$$S = \sum_{i=0}^{8} \frac{8!}{(8-i)!} \tag{4}$$

2. **Uniform cost graph search**. Consider the below state space graph. Perform UCS yourself and provide answers to the questions below regarding the nodes expanded during the search as well as the final path found by the algorithm. Remember that the search procedure should begin at node "START", and the goal state is node "GOAL". To break ties when adding nodes of equal cost to the fringe, follow the alphabetical order.

   Recall that UCS keeps track of the lowest cost, $c(v)$, to get from the start node to the node v.



   (a) What is the order of nodes expanded?
   (b) How many nodes are expanded?
   (c) What is the final path returned?
   (d) What is the length of path?
   (e) What is the cost of path?

   Solution:

   (a) START$\rightarrow p \rightarrow d \rightarrow b \rightarrow e \rightarrow a \rightarrow r \rightarrow f \rightarrow$GOAL
   (b) There are 10 nodes including GOAL node.
   (c) The final path is: START$\rightarrow d \rightarrow e \rightarrow r \rightarrow f \rightarrow$GOAL
   (d) The length of path is 5.
   (e) The total cost of the path is $cost = 3 + 2 + 2 + 2 + 2 = 11$

3. **The graph-coloring problem.** Consider a graph with n nodes that has no multiple edges and loops. We want to color the nodes in this graph. We could only use 2 colors and we need to ensure that there is no edge linking any two nodes of the same color.

   (a) Please judge whether the following graphs could be colored as mentioned above. Write "Y" if you think one graph could be colored as mentioned above and write "N" otherwise.

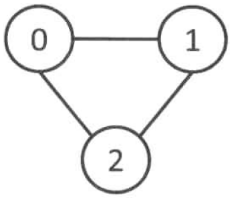


Figure 2: Graph 1.

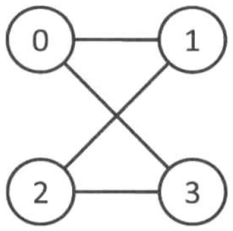


Figure 3: Graph 2.

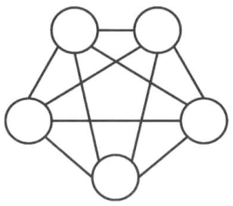


Figure 4: Graph 3.

   Solution: Graph1:N Graph2:Y Graph3:N

   (b) Please briefly describe your findings about the properties of the graphs that could be colored as mentioned above.

   Solution: If the graph is a bipartite graph, it can be colored, otherwise it cannot. Also the above properties is equal to whether there is an odd circle in the graph.

(c) We further assume the graph is connected. Let 1 and -1 be the two color values denoting the two colors respectively. Please fill in the following pseudocode for judging whether a graph could be colored as stated in the problem. (Algorithm 2 returns "True" if a graph could be colored as stated in the problem and returns "False" otherwise)

Algorithm 1 DFS

```
Input: node index $v$, color value $c$, color list $color$
$color[v] \leftarrow c$
for $neighbor$ in get_neighbor_node($v$) do
    if $color[neighbor]$ == __c___ then
        return False
    end if
    if $color[neighbor]$ == 0 and not _____DFS(neighbor,-c,color)______ then
        return False
    end if
end for
return True
```

Algorithm 2 Color the Connected Graph

```
Initialize color list $color \leftarrow (0)_{i=1}^{N}$
if DFS(1,-1,$color$) then
    return __True___
end if
return __False___
```

(d) Please modify Algorithm2 to make Algorithm2 adapt to the graph which might be not connected. (Algorithm 3 returns "True" if a graph could be colored as stated in the problem and returns "False" otherwise)

Algorithm 3 Color the Graph (might be not connected)

```
Initialize color list $color \leftarrow (0)_{i=1}^{N}$
for $node$ in get_all_node() do
    if $color[node]$ == 0 then
        if not DFS($node$,-1,$color$) then
            return __False___
        end if
    end if
end for
return __True___
```

4. **Explore the uniform-cost search in more depth.**

    (a) What are the differences between the uniform-cost search and Dijkstra's algorithm? (You could click here for the reference of Dijkstra's algorithm.)

    (b) If there are some arcs with negative costs in the graph, does the uniform-cost search still work? Give a concrete example to support your argument.

    (c) Introduce one algorithm which you know and is able to find the least-cost path in the graph containing the arcs with negative costs.

    Solution:

    (a) Since Dijkstra's algorithm is used for the graph search, we only consider applying UCS on the graph search. Thus we can propose 3 main difference below:

        i. **Different utility**:Using UCS we can output the shortest distance and corresponding path, but Dijkstra only cares the shortest distance. Using Dijkstra's algorithm cannot help us get the shortest path.

        ii. **Different process**:In Dijkstra's algorithm we only save the distance of each node and terminate the algorithm as having visited all nodes. However, during UCS, we save both the distance and the path and terminate it as reaching the destination.

        iii. **Different applying range**: If there are negative edges in the graph, the UCS will fails. However,even though there exist negative edges, we can still apply Dijkstra's algorithm as long as we guarantee no negative circle.

    (b) UCS will fail! Here is the counter-example Figure 5:

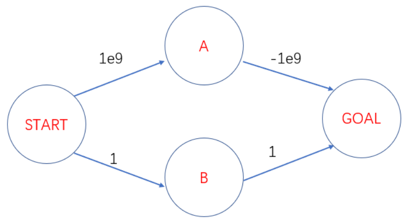


Figure 5: Counter-example

    (c) Bellman-Ford's algorithm for shortest path.

Algorithm 4 Bellman-Ford algorithm

```
1: Input: Graph $G = (V, E)$, Start node $s$, Goal node $g$
2: Output: Shortest path $S$ and distance $D$.
3: for $v \in V$ do
4:     $distance[v] \leftarrow \infty$
5:     $lastnode[v] \leftarrow None$
6: end for
7: $distance[s] \leftarrow \infty$
8: for $i = 1$ to $|V| - 1$ do
9:     for each $e = (u, v) \in E$ do
10:        if $distance[u] + w(e) < distance[v]$ then
11:            $distance[v] = distance[u] + w(e)$
12:            $lastnode[v] = u$
13:        end if
14:    end for
15: end for
16: if $distance[g] == \infty$ then
17:     return FAIL()! #g is not connected to s
18: end if
19: for each $e = (u, v) \in E$ do
20:     if $distance[u] + w(e) < distance[v]$ then
21:         return FAIL()! #negative circle.
22:     end if
23: end for
24: $S \leftarrow NONE$
25: $seach\_node \leftarrow g$
26: while $seach\_node \neq s$ do
27:     $S.insert(0, seach\_node)$
28:     $seach\_node = lastnode[seach\_node]$
29: end while
30: return $S$,$distance[g]$
```