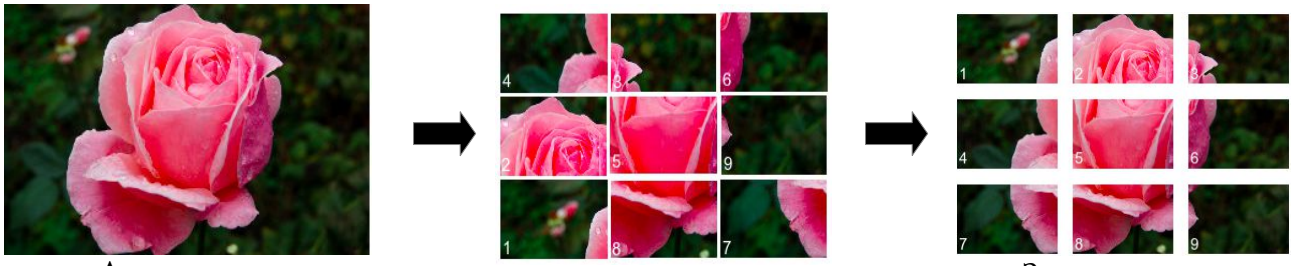
Лабораторын ажил 3

Оюутан бүр зураг эвлүүлэх puzzle тооцооллыг хийнэ үү.

1. Сонгон авсан зургийг 9 хэсэгт хуваах буюу 3x3 хэмжээстээр дүрслээд дүрснүүдийг санамсаргүйгээр холино үү.
2. Холисон дүрснүүдийг хайлтын мод ашиглан эвлүүлнэ үү. Холисон төлөвт байгаа зургийн puzzle-ийг зорилгын төлөвт хүрэх замын өртөгийг тодорхойлно уу. Мөн хугацаа, санах ойн хувьд зарцуулалтыг тооцоолно уу.



Анхны төлөв Холисон төлөв Зорилгын төлөв

Холисон төлөвийн puzzle нь санамсаргүйгээр оноогдох тулд тооцоологдох өртөг нь харилцан адилгүй байна. Хайлтын алгоритмуудаас ашиглана уу. Дараах алгоритмуудаас 2-ийг сонгож үр дүнг харьцуулан үзэж болно. Үүнд:

- Iterative deepening search
- Bidirectional search
- Alpha, beta prunning search
- Breadth-First Search
- Uniform cost search
- Depth first search
- Depth limited search
- A* search

Ашиглаж болох функцүүдийн Pseudo code-ын загваруудыг доор үзүүлэв.

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

```
DFS(G,v)   ( v is the vertex where the search starts )
         Stack S := {};   ( start with an empty stack )
           for each vertex u, set visited[u] := false;
                          push S, v;
                    while (S is not empty) do
                            u := pop S;
                      if (not visited[u]) then
                          visited[u] := true;
                  for each unvisited neighbour w of u
                              push S, w;
                            end if
                          end while
                        END DFS()
```

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```