**Artificial Intelligence**

**Homework 1 [24 pts total]**

**Graph Search**

**Elham Sadeghi**

**Given the undirected graph G1 represented as vertex-list:**

**Perform DFS using stack for search and function call recursion.**

* the search state expanded: s, d ,b ,a ,c ,f ,G
* the search path returned: s , d , b , a , c , f , G

**A screen shot of a computer

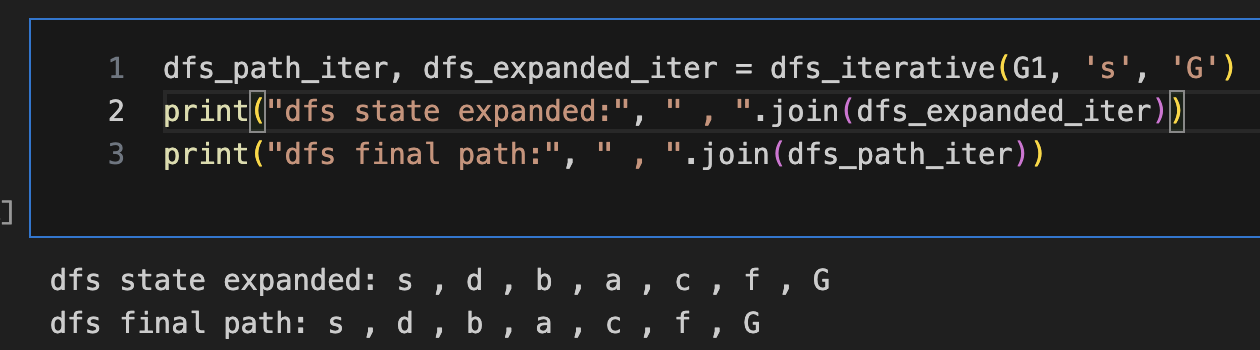
Description automatically generated**

In my implementation, I used:

* sorted(): A built-in function to sort the neighbors of each node before exploring them.
* set(): Used to keep track of visited nodes to avoid revisiting as we cannot have repetition in set.
* list(): Used for tracking the path and the expanded nodes during the search.

**Perform DFS using stack for search and (stack + while loop) to replace recursion calls.**

* the search state expanded: s , d , b , a , c , f , G
* the search path returned: s , d , b , a , c , f , G

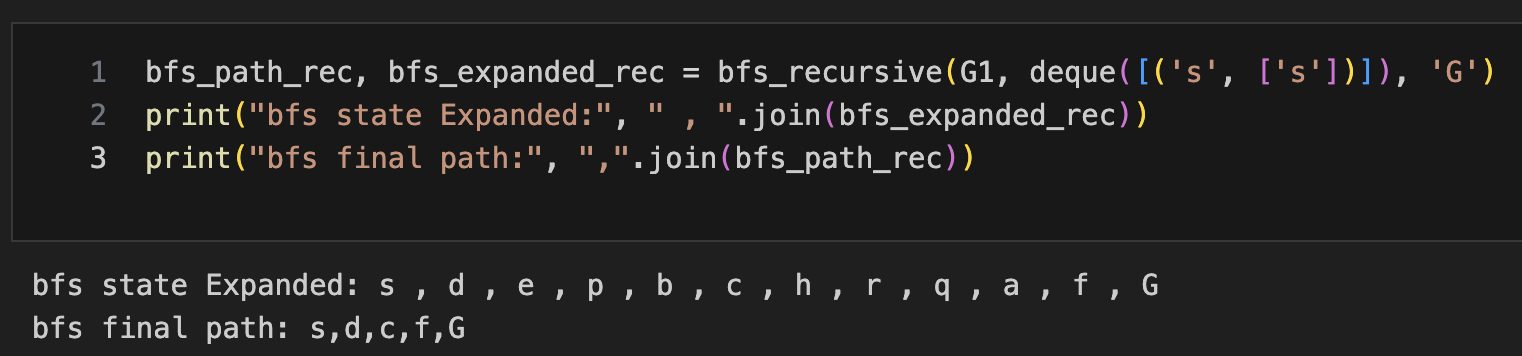
****

In my implementation, I used:

* sorted(): A built-in function to sort the neighbors of each node in reverse order before exploring them.
* set(): Used to keep track of visited nodes to avoid revisiting.
* list(): Used for tracking the path and the expanded nodes during the search.
* stack: A list used as a stack to manually control the nodes to be explored.
* while: A loop used to continue the search while there are nodes left in the stack.

**Perform BFS using queue for search and function call recursion.**

* the search state expanded: s , d , e , p , b , c , h , r , q , a , f , G
* the search path returned: s,d,c,f,G

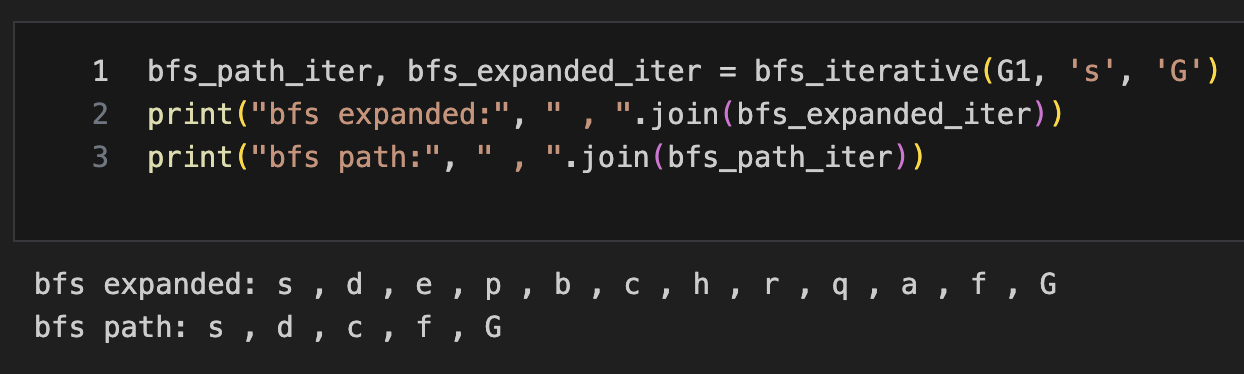
****

In my implementation, I used:

* sorted()
* set()
* list()
* queue: A double-ended queue (deque()) to manage nodes in the bfs order.
* popleft(): A method used to remove nodes from the front of the queue for processing.

**Perform BFS using queue for search and (stack + while loop) to replace recursion calls.**

* the search state expanded: s , d , e , p , b , c , h , r , q , a , f , G
* the search path returned: s , d , c , f , G

****

In my implementation, I used:

* sorted()
* set()
* deque()
* popleft()
* list()

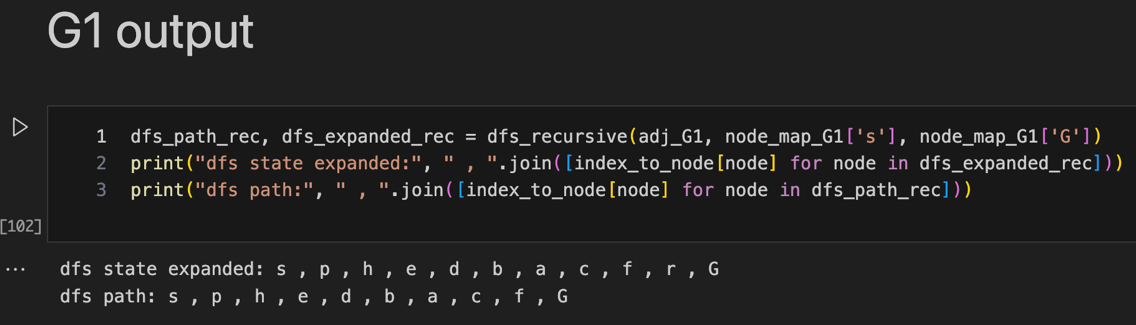
A whiteboard with writing on it

Description automatically generated

**Given the undirected graph G1 represented as adjacency matrix:**

**Perform DFS using stack for search and function call recursion. (1pt)**

* the search state expanded: s, p , h , e , d , b , a , c , f , r , G
* the search path returned: s , p , h , e , d , b , a , c , f , G

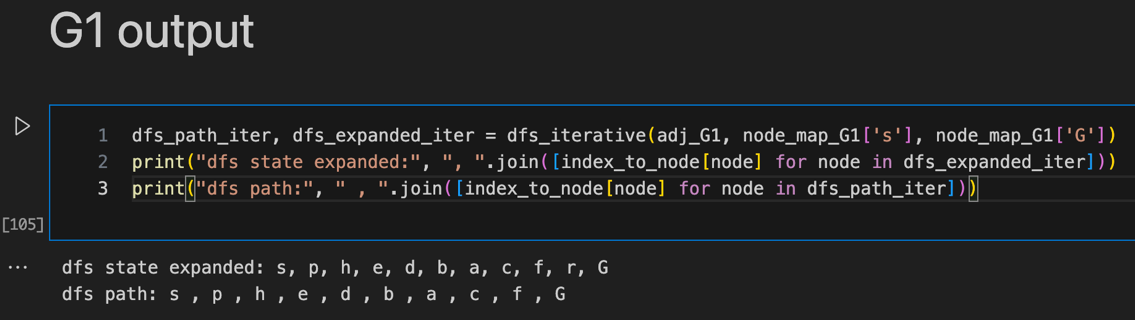
****

In my implementation, I used:

* enumerate(): A built-in function used to loop through the neighbors of the current node and check if they are connected.
* list(): Used for tracking the path of visited nodes and the order in which nodes are expanded during the search.
* recursion: The function calls itself to explore the graph deeply, simulating the behavior of dfs.

**Perform DFS using stack for search and (stack + while loop) to replace recursion calls.**

* the search state expanded: s, p, h, e, d, b, a, c, f, r, G
* the search path returned: s , p , h , e , d , b , a , c , f , G

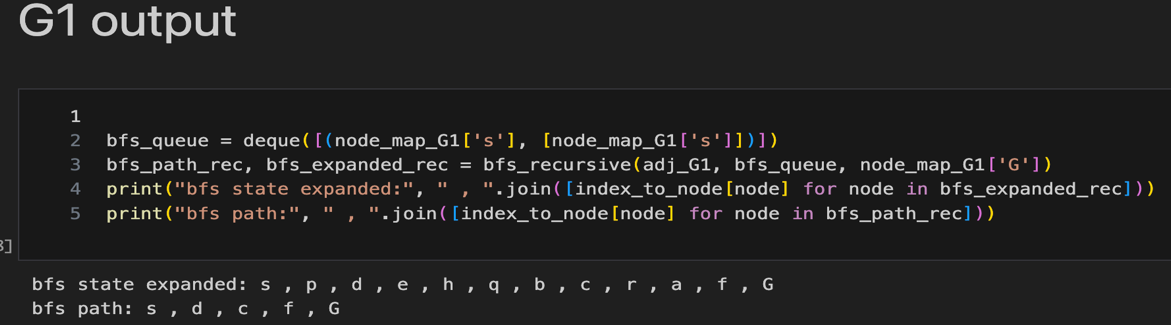
****

In my implementation, I used:

* set()
* stack
* pop(): A method used to retrieve and remove the last element from the stack
* range(): A built-in function used to loop through the neighbors in reverse order to ensure the correct order of traversal.
* list()

**Perform BFS using queue for search and function call recursion. (1pt)**

* the search state expanded: s, p , d , e , h , q , b , c , r , a , f , G
* the search path returned: s , d , c , f , G

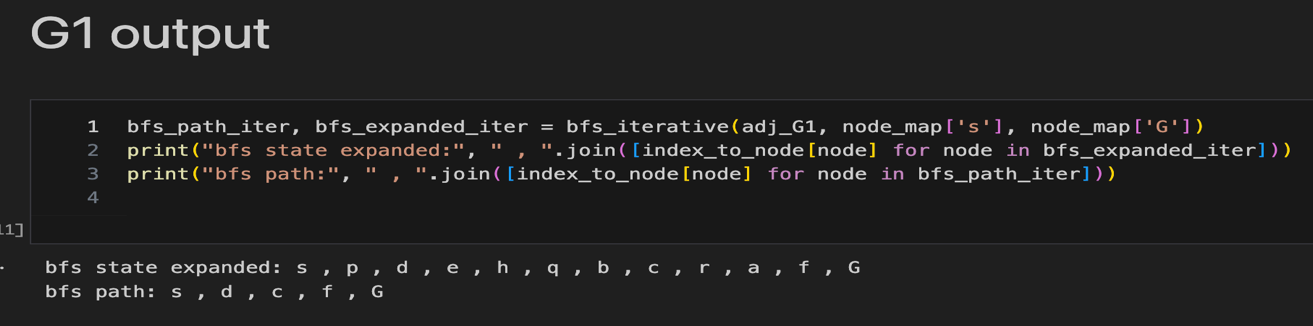
****

In my implementation, I used:

* enumerate()
* set()
* deque.popleft()
* deque.append(): Used to add nodes to the queue for exploration.
* recursion: The function calls itself to explore nodes in a breadth-first manner.
* list()

**Perform BFS using queue for search and (stack + while loop) to replace recursion calls.**

* the search state expanded: s , p , d , e , h , q , b , c , r , a , f , G
* the search path returned: s , d , c , f , G

****

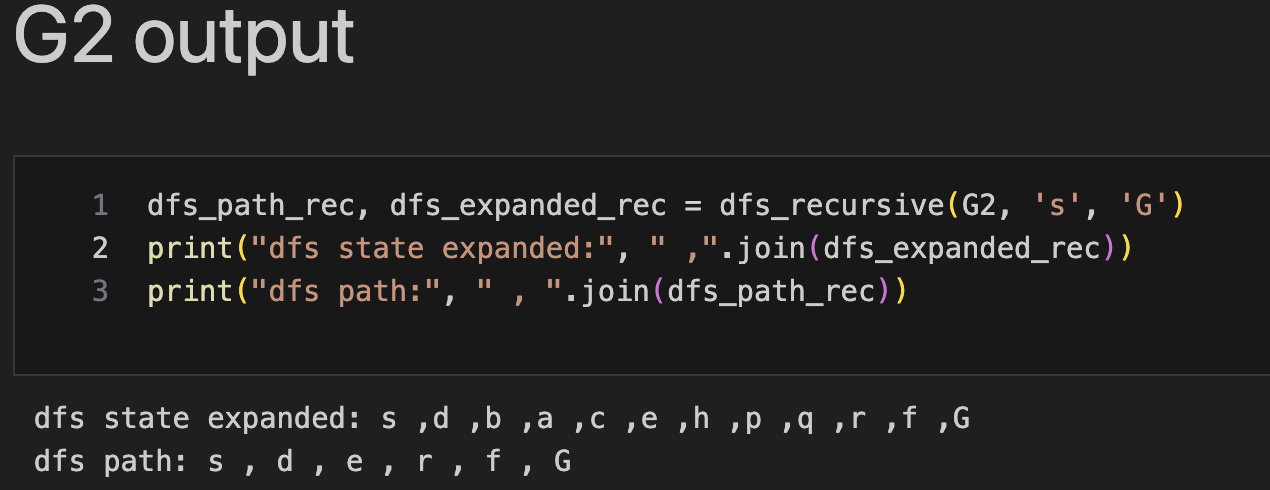
In my implementation, I used:

* set()
* deque.popleft()
* deque.append()
* range(): A built-in function used to loop through the neighbors of each node.
* list()

**Given the directed graph G2 represented as vertex-list:**

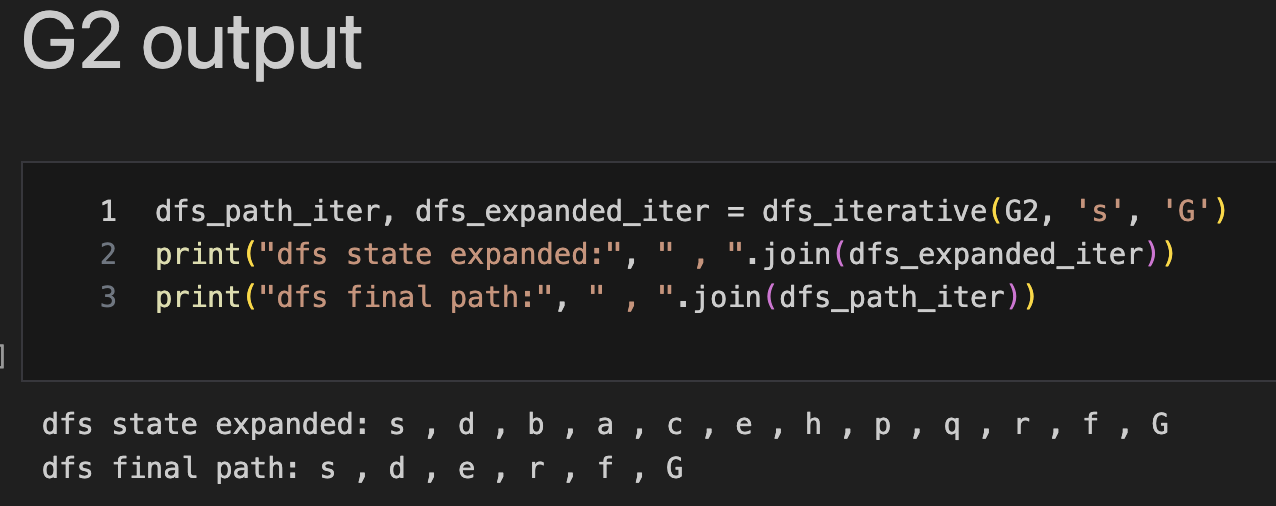
**Perform DFS using stack for search and function call recursion. (1pt)**

* the search state expanded: s ,d ,b ,a ,c ,e ,h ,p ,q ,r ,f ,G
* the search path returned: s , d , e , r , f , G

****

**Perform DFS using stack for search and (stack + while loop) to replace recursion calls.**

* the search state expanded: s , d , b , a , c , e , h , p , q , r , f , G
* the search path returned: s , d , e , r , f , G

****

**Perform BFS using queue for search and function call recursion. (1pt)**

* the search state expanded: s , d , e , p , b , c , h , r , q , a , f , G
* the search path returned: s,e,r,f,G

A computer screen shot of a code

Description automatically generated

**Perform BFS using queue for search and (stack + while loop) to replace recursion calls.**

* the search state expanded: s , d , e , p , b , c , h , r , q , a , f , G
* the search path returned: s , e , r , f , G
* **A screen shot of a computer code

  Description automatically generated**

**A screenshot of a cell phone

Description automatically generated**

**Given the directed graph G2 represented as adjacency matrix:**

**Perform DFS using stack for search and function call recursion. (1pt)**

* the search state expanded: s , p , d , a , h , f , q , f , r , p , d , d , b , e , c , q , a , h , f , r , p , d , d , f , G
* the search path returned: s , b , e , c , G

**A screen shot of a computer

Description automatically generated**

**Perform DFS using stack for search and (stack + while loop) to replace recursion calls.**

* the search state expanded: s, p, d, a, h, f, q, r, b, e, c, G
* the search path returned: s , b , e , c , G

**A computer screen shot of a code

Description automatically generated**

**Perform BFS using queue for search and function call recursion.**

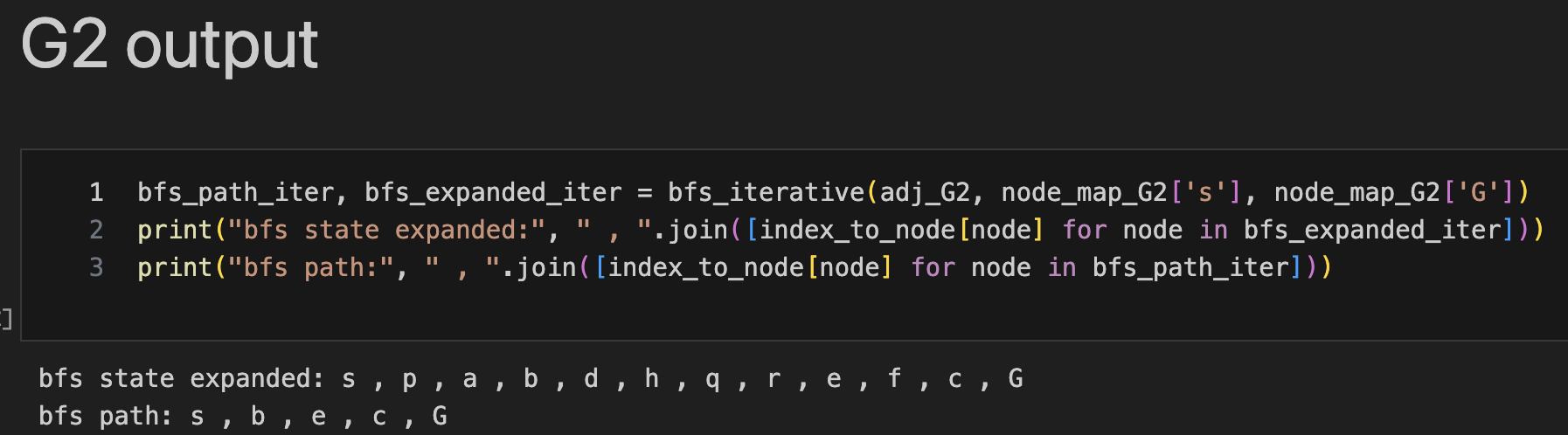
* the search state expanded: s , p , a , b , d , h , q , r , e , f , c , G
* the search path returned: s , b , e , c , G

**A computer screen shot of a code

Description automatically generated**

**Perform BFS using queue for search and (stack + while loop) to replace recursion calls.**

* the search state expanded: s , p , a , b , d , h , q , r , e , f , c , G
* the search path returned: s , b , e , c , G

****

**Given the undirected weighted graph G3 represented as vertex-list:**

**Perform UCS using PQ. (2pt)**

* UCS Search State Expanded: s , p , d , b , e , h , a , r , c , f , q , G
* UCS Path Returned: s , d , e , r , f , G
* Total Cost (Vertex List): 11

**Both dfs and bfs can produce different orders of expanded nodes depending on the order in which neighbors are processed and the representation used (vertex list or. adjacency matrix). For instance, DFS may explore neighbors in a different order based on how they are stored or sorted.**

**A computer screen shot of a program code

Description automatically generated**

In my implementation, I used:

* heapq.heappop(): Used to remove and return the node with the smallest cost from the priority queue.
* heapq.heappush(): Used to add neighbors to the priority queue with their corresponding cost.
* set(): Used to track visited nodes and prevent revisiting.
* list(): Used for tracking the path of visited nodes and the order in which nodes are expanded.
* sorted(): Used to ensure the neighbors of the current node are processed in a consistent order.
* I did not use any external python modules in this implementation, apart from the heapq module for the priority queue.

**Given the undirected graph G3 represented as adjacency matrix:**

**Perform UCS using PQ. (2pt)**

* UCS Search State Expanded (Adjacency Matrix): s , p , d , b , e , h , a , r , c , f , q , G
* UCS Path Returned: s , d , e , r , f , G
* Total Cost: 11

**A computer screen shot of text

Description automatically generated**

In my implementation, I used:

* heapq.heappop()
* heapq.heappush()
* set()
* enumerate()
* list()

**A diagram of a tree

Description automatically generated**

**Given the directed weighted graph G4 represented as vertex-list:**

**Perform UCS using PQ. (2pt)**

* UCS Search State Expanded (Vertex List): s , p , d , b , e , a , r , f , G
* UCS Path Returned: s , d , e , r , f , G
* Total Cost: 11

**A computer screen with text and symbols

Description automatically generated**

**Given the directed weighted graph G4 represented as adjacency matrix:**

**Perform UCS using PQ. (2pt)**

* UCS Search State Expanded (Adjacency Matrix): s , p , d , b , e , a , r , f , c , G
* UCS Path Returned: s , d , e , r , f , G
* Total Cost: 11

**A screen shot of a computer code

Description automatically generated**

**A diagram of a tree

Description automatically generated**

**Based on my implementation and experience, I found that the representation of the graph  (vertex list or adjacency matrix) might affect the order of node expansion in UCS. but, this does not impact the final path found because UCS prioritizes nodes based on their accumulated cost. UCS will always find the optimal (least-cost) path, regardless of the representation, but the order in which nodes are expanded during the search may vary.**