# Artificial Intelligence - CS6601 - Assignment One - Search
# Submitted by Deepak Edakkattil Gopinath (903014581)

## Warmup

1. IDS can be worse than depth first search when the branching factor is really high and the goal node is at a very deep level but finite depth. This is because for IDS, if the goal node is at a very deep level, the tree structure before the goal depth has to regenerated everytime the iteration happens. This is wasteful in terms of memory complexity. Whereas in Depth First Search, the nodes are only created once and the traversal happens along a narrow path along the side.

2. a) BFS is a special case of uniform cost search when the step costs are all the same
b) In Best First Search a new node is chosen for expansion based on an evaluation function. UCS is a special case of Best First Search when the evaluation function is g(n). Greedy Best First Search is one in which the evaluation is h(n). Best First Search becomes a Breadth First Search when the when f(n) = constant.
We can construct a tree such that b = 2. And
f(n) = -d-1 (for a node n at depth d and on branch 1) and f(n) = -d (for a node n at depth d and on branch 2)
In this example, for best first tree search the tree would be traversed along branch one at all depth as the nodes on branch 1 would have the lowest f(n) for any depth d. And this would be similar to depth first search.
c) Uniform Cost Search is a special case of the A* search when the heuristic function h(n) is 0 for all nodes.

## Practice

1. This has to do with the memory allocation and element swapping that happens a lot more when we try to sort a regular array. Whereas in the case of a priority queue the element swapping is very minimal.

For fibonacci heap the insertion complexity is O(1), whereas for deleting the minimum (for popping) it is O(log n).

For unordered arrays and sorting depending on the sorting algorithm the time complexity can be O(n^2) or O(nlogn)

Therefore using a fibonacci heap or other forms of priority heaps will have a significant advantage over regular arrays.

2. I don’t know.

**Implementation**

Three separate algorithms were implemented. As the question requires us to find the shortest path (the optimal path with the shortest path cost) a uniform cost search version of the bidirectional search was implemented. Because only then would the algorithm guarantee an optimal path, at least theoretical.
The tridirectional search is treated almost like having 3 parallel Bidirectional searches running at the same time. The advantage of tridirectional search over three bidirectional searches in parallel is that the number of times the frontiers are expanded is much less, thereby saving time and memory.

In uniform cost search a dictionary is used to keep track of the distances of the states from the start. If a node is generated with a lower path cost then the dictionary is updated accordingly to reflect the new lower path cost. The goal test is done right after a node is popped off the priority queue.

The *heapq module* available in Python is used for the priority queue. A euclidean distance measure calculated from the latitude and longitude of the locations are used for calculating the step costs and the path costs. A helper function find_euclidean_distance() was written. The path cost to a state is used as the priority measure for sorting the queue.

Issues with bidirectional uniform cost search:

In bidirectional search the frontier expansion is typically stopped when an intersection is found. The intersection check is usually done between the last popped node from one frontier and the nodes in the newly formed frontier of the other and vice-versa and also between the two popped nodes. If an intersection is found, then at this stage at least one of the semi-paths is guaranteed to be optimal as the node was just popped off from the frontier.  In order to guarantee optimality for the other “half” of the path, I decided to continue expanding that frontier until the node that was part of the matched pair is eventually popped out of the frontier. This way we can guarantee optimal path for the other “half” as well.

This is made possible by keeping flags for whether the forward and backward search should be on or not.

In tridirectional search, all three frontiers are expanded simultaneously. The goal is to find the intersection points between any pair of cities. Unlike the bidirectional search even if a popped node (say from frontierA) finds a matching node with a node in frontierB, we might still have to continue expanding frontierA until a matching node is found between A and C. The three frontiers stop only when all the three paths have been obtained. The implementation is just like bidirectional, with additional flags to make sure that the frontier expansion is continued until optimal paths are found from the intersection states. The intersection nodes are stored in a list.

Uniform Cost Search by definition, is optimal and completeness is also guaranteed if the path cost is greater than zero (which is true in this case).
Bidirectional Uniform Cost Search also theoretically is optimal and complete. When I compared the path cost from bidirectional search and uniform cost search, there were many instances where the results were exactly the same, whereas many times the path costs were different. The time taken and the nodes expanded for the BDUCS can be higher than a regular a UCS because the frontier is not necessarily expanded out symmetrically.
Tridirectional search: The way I have tried to implement, is based on BDUCS, and I expect it to be complete and optimal. But just like bidirectional search I am having issues with total path costs. There were a few instances when all three search methods returned the same exact path costs. I am aware of some termination condition issues in my tridirectional search, because when i printed out the backtracked nodes, there were some duplications.

## **Evaluation**

*Experimental setup:*
Three cities are chosen at random.
3 separate UCS are done on each pair of cities. Let the costs be costAB, costAC, costBC.
I calculate pairwise sum of the costs and the lowest sum pair represents the most optimal path.
The same is repeated when 3 separate BDS are run and the lowest sum for each pair of path costs represents the most optimal path.
The tridirectional search is done once and the path costs are computed in a similar way.

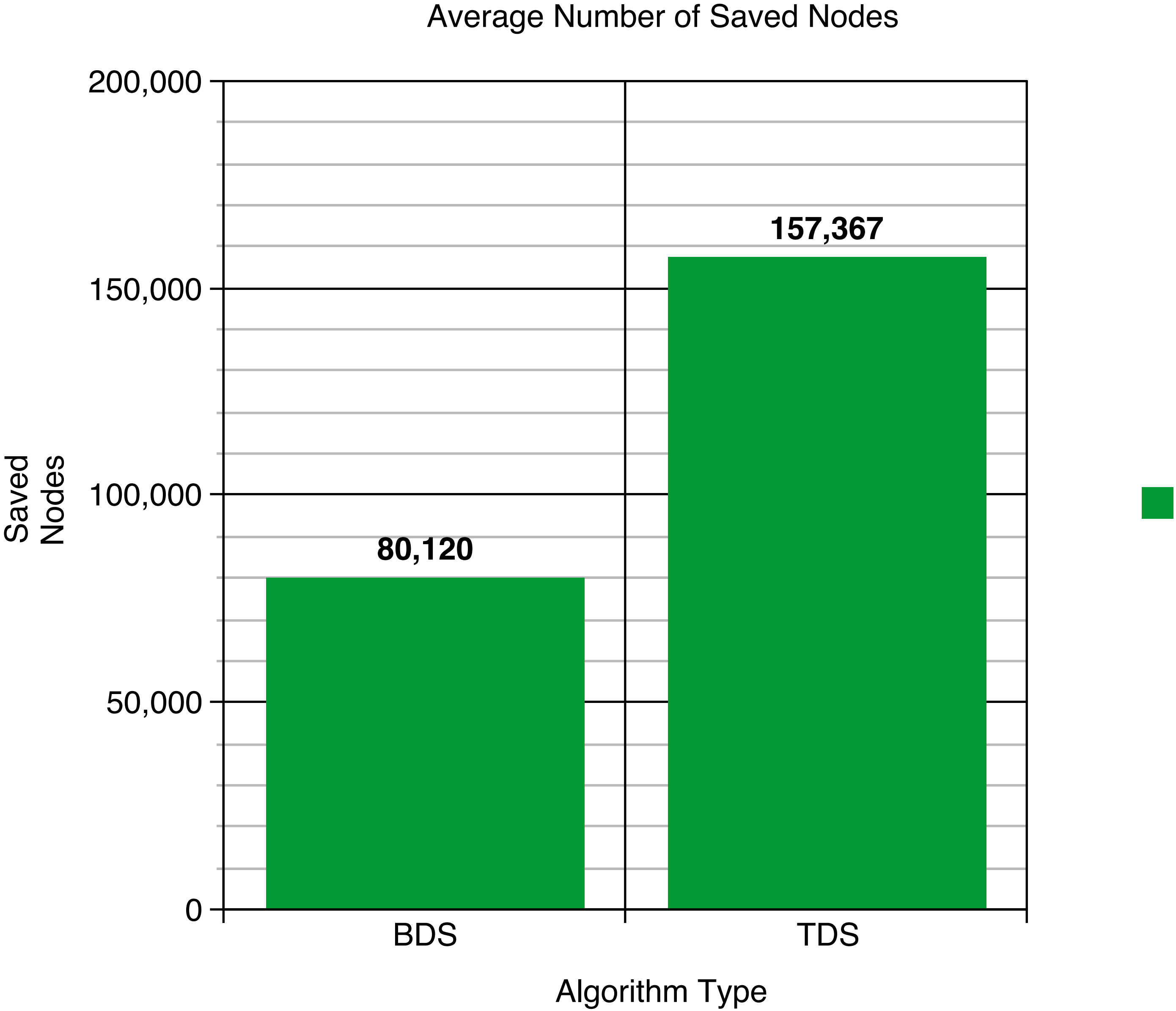
In general the number of nodes(UCS) > nodes(BDS) > nodes(TDS). In some cases the TDS would return a wrong path and I have not been able to figure out exactly what is causing this issue. I know that there are a few duplicated nodes in some of the paths in TDS.
There were numerous cases in which the no solutions were found for a pair. I think this might have to do with the nature of the graph itself. There might be some unconnected sections in the graph.

The total number of nodes explored for 100 iterations:

**1. Uniform Cost Search - 23,477,350**
**2. Bidirectional Uniform Cost Search - 15,465,301**
**3. Tridirectional Uniform Cost Search - 7,740,662**

Average Number of Saved Nodes

**1. Bidirectional UCS = (23477350-15465301)/100 = 80120.49 ~ 80120**
**2. Tridirectional Uniform Cost Search (23477350 - 7740662)/100 = 157366.88 ~ 157367**


Average Number of Saved Nodes
Saved Nodes
200,000
150,000
100,000
50,000
0
157,367
80,120
BDS
TDS
Algorithm Type