**Periphery Sweep Algorithm: Conquering A\* algorithm at graph traversal solutions**

1. **Introduction**
	1. Pathfinding is the process of using a computer application to find and plot the shortest route between two points. Pathfinding is closely associated with the shortest path problem which examines how to identify the path that best meets some criteria (shortest, cheapest, fastest, etc.) between two points in a large network.
	2. The field of pathfinding is heavily based on Dijkstra’s algorithm for shortest path problems developed by Edsger Dijkstra in 1956.
	3. Pathfinding finds a huge spectrum of applications in robotics, route-planning and game development. It is also the basis of working of Google Maps.
	4. In computer science, pathfinding is accomplished through graph traversal algorithms. These algorithms specify a particular order in which nodes are to be searched from the source node in order to reach the destination node best meeting some weight associated criterion.
	5. The current industry standard for pathfinding applications is the A\* graph traversal algorithm which is an intelligent version of Dijkstra’s algorithm. It considers a simple estimate from the source node to the destination node, called the heuristic, to traverse through the graph.
	6. This paper introduces Periphery Sweep, a new algorithm which draws its motivation from IDA\* or iterative deepening A\*, a space efficient variant of A\*. IDA\*, by itself, is an extremely slow and inefficient graph traversal algorithm which yields worse results than the A\* algorithm. Periphery Sweep introduces some key changes to tackle the inefficiencies of IDA\* algorithm in an attempt to provide a faster graph searching solution than A\*.
2. **Objective:**
	1. A\* algorithm utilizes a data structure called the priority queue which is the key to making greedy decisions necessary for finding the path of least cost from source node to destination node. While the priority queue allows A\* algorithm to expand nodes in a best-first manner, it yields a time complexity of O(log N), where N is the number of elements in the priority queue, for insertion and removal operations. As the search space becomes large, it becomes increasingly expensive to maintain the priority queue. Periphery Search tries to achieve the greedy best-first approach without using the priority queue. This helps in introducing the scope of relatively more efficient data structures for just keeping track of nodes to be explored.
	2. Periphery Sweep makes some radical changes in IDA\* to reduce the search space and avoid repetitive work.
3. **Underlying principles and motivation**

The discussion of the inefficiencies of A\* vs IDA\* would clear the way for understanding the need for Periphery Sweep algorithm.

* 1. Inefficiencies of IDA\* algorithm:
		1. **Explores unnecessary multiple paths to the destination node**

Since IDA\* uses iterative deepening without any form of storage, it ends up exploring multiple paths(or the same nodes) on its way from the source node to the destination node. This results in a lot of unnecessary work that isn’t essential for finding the best path.

A\* does not suffer from this problem because it maintains an open list using the priority queue to store information about the best path at a specific stage. It also maintains a closed list through a set to check for already visited nodes.

* + 1. **Repeats all the work of previous search cycle**

 IDA\* uses depth first search to construct the search boundary (the boundary of a search cycle consisting of all the nodes to be explored). However, for every (i+1)th search cycle, IDA\* must start from the source node and perform depth first search all over from the start node to construct the new search boundary which just happen to be the children of the search boundary in case of ith search cycle. So, (i+1)th search cycle must repeat everything that ith search cycle performed.

 This is by far the most expensive process performed by IDA\*.

A\* maintains the search boundary in paths stored inside the priority queue. These paths may be a vector of nodes or node pointers. So, we can easily know the last nodes that were visited for each vector(or path). As a result, A\* can continue its search from where it was last left.

 **3.1.3** **The search boundary is traversed from left to right**

 All the nodes contained in the search boundary are explored from left to right in iterative deepening. IDA\* does not necessarily prioritize the node that was achieved with the least cost. Instead, it checks all the nodes in the search boundary from left to right. Although nodes are expanded in best-first manner in IDA\*, nodes are explored from left to right.

 A\*, however, explores and expands nodes in a best-first manner due to the priority queue. Therefore, nodes which look most promising to lead to the destination node are explored first.

* 1. Inefficiencies of A\* algorithm:
		1. **Priority queue:**

It becomes increasingly expensive to maintain the priority queue in case of A\* algorithm as the number of nodes in search space increase because of insertion and removal operations being O(log N).

1. **Implementation details :**

From the points above, it can be clearly seen that if we are able to remove the inefficiencies of IDA\* as discussed in point 3.1.1 and 3.1.2, we can gain a significant performance gain over A\*.

 Removing the inefficiency in point 3.1.3 is difficult because if we try to prioritize exploration of certain nodes over others in case of IDA\*, we must use a priority queue which would defeat the entire purpose of the algorithm. It is to be noted that although it is difficult to explore nodes in a best-first manner, expansion of nodes occurs in the exact same manner as A\*. This process of not exploring the best nodes first is not very costly.

* 1. The first inefficiency of IDA\* is removed by maintaining a hash map which maps from ***the pointer to a node*** to the ***path cost till that node***. The map can be searched for checking if a specific node during a search cycle has already been attained through a path with lesser cost. If the node is achieved by a better path, then the hash map is updated with the new value of path cost for that node. Otherwise, that node is skipped, and no exploration is done. Another hash map is used which maps from ***the pointer to a node*** to the ***pointer to the parent of that node.*** It is updated with the new parent node through which the given node is achieved to keep track of the path. The utility of the second hash map will be clearer from the next point.
	2. The second inefficiency of IDA\* is removed by maintaining 2 data structures. A doubly linked list is used to keep track of the nodes in the search boundary(or periphery). The algorithm is designed in such a way that there is absolutely no need to keep track of the parent of a particular node in the search boundary. Every time a node is explored, and its children expanded, its parent is removed from the search periphery. This helps in avoiding the work of the previous cycle. However, since we need to be aware of how we reached the destination node when we actually reach it, we need to maintain a hash map which maps from pointer to a node to the pointer to its parent node. This was the utility of the second hash map in the previous point.

**Note:** The fact that we are removing the parent completely from the search space is the most radical change incorporated in the Periphery Sweep algorithm. So, the algorithm can be seen as a search periphery(or boundary) or even a wave which sweeps and keeps on spreading out eventually until the wave eventually hits the destination node. At a certain stage in the search, we only keep track of the nodes in the search periphery and absolutely forget about how these nodes were arrived at. Hence, the algorithm is named Periphery Sweep.

* 1. The Periphery Sweep algorithm requires only 3 data structures to work but another hash map has been added for optimizing the removal operation from the doubly linked list which maintains the nodes in the search periphery. Since the need for priority queue has been eliminated through iterative deepening, we can make our insertion and removal operations to be as efficient as possible. This hash map maps from a ***pointer to a node*** to an ***iterator in the doubly linked list*.** The overall result of maintaining this additional data structure is that we get O(1) complexity for insertion and removal operations.
1. **Results :**
	1. A fairly optimized version of A\* algorithm was tested against Periphery Sweep algorithm and it was found that Periphery Sweep offered a performance gain of **at least 40%** over A\* algorithm for similar inputs in graph testbeds. The performance gain improves drastically as the number of nodes in the search space increases.
2. **Future scope:**
	1. While it is difficult to completely optimize the third inefficiency of exploring the search periphery in a best-first manner, there can be some partial sorting algorithm which helps us to explore those nodes first which look most promising to yield the shortest path to the destination node. However, we need to make sure that the sorting process is better than O(log N).
3. **C++ Implementation:**

****

1. **Time taken and IDE (Integrated Development Environment) used:** 4 months and Qt Creator