Graph and Priority Queue implementation:

Priority Queue

```
class Astar_Node:
    def __init__(self, heuristic, index = None):
        self.index = index
        self.heuristic = heuristic
        self.priority = None
        self.next = None

class Astar_PriorityQueue:
    def __init__(self):
        self.front = None

    def isEmpty(self):
        if self.front == None:
            return True
        else:
            return False

    def push(self, cost, node:Astar_Node):

        node.priority = node.heuristic + cost

        if self.isEmpty() == True:
            self.front = node
            return

        else:
            if  node.priority < self.front.priority:
                node.next = self.front
                self.front = node

            else:

                ptr = self.front
                while ptr:

                    if ptr.priority<= self.front.priority:
                        break
```

```
            ptr = ptr.next

        node.next = ptr.next
        ptr.next = node

    return

def pop(self):
    if self.isEmpty():
        return None
    else:
        front_node = self.front
        self.front = self.front.next
        return front_node

def peek(self):
    if self.isEmpty():
        return None
    else:
        return self.front.heuristic
```

Graph Implementation:

```
from AStar_PriorityQueue import Astar_Node
class Astargraph:
    def init(self, num):
        self.x = num
        self.graph = [[] for i in range(self.x)]

    def edge_add(self, x, y, weight):
        node = Astar_Node(index=x)
        self.graph[y].append((node, weight))

        node = Astar_Node(index=y)
        self.graph[x].append((node, weight))
```

Repeated Astar and Backwards repeated implementation

*Repeated A**

```
frontier = AStar_PriorityQueue.Astar_PriorityQueue()
    cost = 0
    frontier.push(cost, start)
    path = []
    while frontier:
        current = frontier.pop()  # Node Type
        path.append(current)
        cost += g.graph[current.index][1]

        if current == goal:
            break

        for neighbors in range(g.x):
            temp = g.graph[current.index][neighbors]
            frontier.push(temp, cost)
```

*Backwards Repeated A**

```
path = []
while start != goal:
    # Repeated A* Code
    frontier = AStar_PriorityQueue.Astar_PriorityQueue()
    cost = 0
    frontier.push(cost, start)
    path = []
    while frontier:
        current = frontier.pop()  # Node Type
        path.append(current)
        cost += g.graph[current.index][1]

        if current == goal:
            break

        for neighbors in range(g.x):
            temp = g.graph[current.index][neighbors]
            frontier.push(temp, cost)

    # Backwards A-Star
```

```
for i in range(len(path) - 1):
    print(path[i])
```

Runtime Conclusions:

When comparing runtimes at different start points and goal points the runtime varies by proximity. It seems that when the start cell is deeper within gridworld or graph backwards A* is more efficient as it is closer to the end and generally has a faster runtime when it first initiates the start. However, when the starting cell is closer/less deep in the gridworld or graph then A* is more efficient.

These phenomena occur because the search algorithms branch outwards and are generally quicker when they are closer to the start as they incorporate less nodes/grids