**Problem Statement:**

The IEEE 802.15 mission 6 is attempting to build an efficient power saving and low recurrence short coverage correspondence usual convention for Body Area Network (BAN).

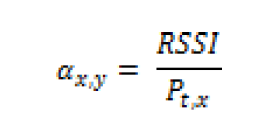
Our project is to adapt modified Dijkstra Algorithm utilising the link cost function to reduce the energy consumption in the WBAN .

Approach:

* Background Study.
* Why do we use Dijkstra Algorithm as an energy efficient routing algorithm?
* Working of Dijkstra Algorithm.
* Implementation.
* Performance comparison.
* Conclusion.

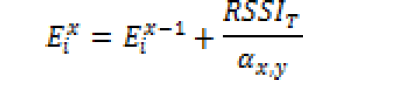
**Background study:**

In the proposed algorithm, the collected vitality utilized by every hub is calculated in too. Connection cost data is occasionally accumulated at the AP(Access Point) as channel constriction for each connection in the system. The channel attenuation for the chosen connect between hub x and hub y, αx, y .

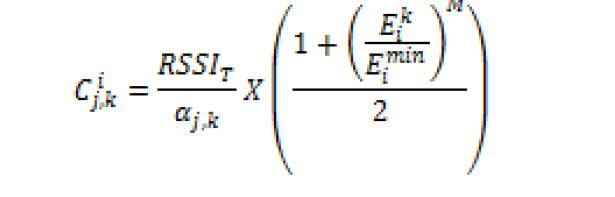


Received Signal Strength Indicator (RSSI) is measured by hub y and Pt, x is the transmitted power utilized by hub x.

For each round of iteration, information is collected called connection state information, every hub's standardized vitality utilized up to this point



The rate of the connection between the hubs i and j is ascertained as



The rate factor in terms of energy is obtained by dividing the aggregated energy utilized by the receiver hub, Ey, with the base collected energy over all hubs .

This proportion is then raised to the energy of M ≥ 0, which decides how tough the impact of vitality irregularity would be.

In the event that the vitality spent by a particular hub is substantially more noteworthy than the present least, it will probably be stayed away from as a hand-off for different hubs, since its approaching connection rate in terms of energy is high.

The cost issue is standardized in the Equation 3 when the variable M = 0 it diminishes to the regular cost work which is the energy required to transverse the connection paying little mind to aggregated vitality crosswise over hubs in the system.

**Why do we use Dijkstra Algorithm as an energy efficient routing algorithm?**

The proposed method leverages Dijkstra's Algorithm, traditionally used for finding the shortest paths in graphs, to prioritize routes based on energy efficiency. The Link Cost Function is introduced to evaluate the energy state of nodes, enabling the algorithm to select paths that balance energy consumption across the network. Key features of the approach include:

* **Energy-Aware Routing:** The algorithm takes into account the remaining energy of sensor nodes, prioritizing paths that maximize energy efficiency and minimize depletion.
* **Improved Packet Delivery:** By optimizing routes based on energy levels, the algorithm enhances the likelihood of successful packet delivery, particularly in dynamic environments.
* **Reduced End-to-End Delay:** The efficient selection of paths leads to lower delays, making the network more responsive to real-time monitoring requirements.
* **Sustainability:** By distributing energy usage evenly among nodes, the algorithm extends the overall network lifetime, reducing maintenance and operational challenges.

**Working of Dijkstra Algorithm:**

**Step 1:** initialize all the nodes as unvisited, node to all other nodes distance is ∞ and nodes previous is the same node.

**Step 2:** distance to access point is 0 and find the minimum node energy in all nodes

**Step 3:** if any unvisited nodes available then find the node with smallest distance and assign to nodesource as a source node and perform the following for all the links of source

a) If link of a source is equal to nodesource then nodedest is equal to link destination and nodedest is not visited then find cost ← link power \* (1 + (nodedest energy/mini energy)M)/2 and new distance of the node is nodesource.distance + cost

b) if distance is less the nodedest distance then nodedest distance is become new distance and node previous is the source node. nodesource.Marked visited

**Step 4:** repeats step3 until all nodes visit.

Dijkstra's calculation is a base up work for making a system tree by for fear that vitality overheads course over all directing ways in the system. The algorithm finds a different network tree for each run/each routing.

**Implementation :**

A prototype has been created using python network simulator modules such networkx and plotting tools such as matplotlib , animating modules to showcase the routing of data packet

Graphs are made for the analyzing of fluctuation in the energy level of the sensor nodes during each routing rounds.

Using Proxy servers in the routing increases the novelity where proxy servers stores the optimal path during each round of routing and helps for the better and time efficient routing for the next time.

Using popular network simulators like ns-3 and omnet++ , we try to demonstrate the working and using graphical analyzing tools the performance metrics are analysed.

**Python code:**

import random

import matplotlib.pyplot as plt

import networkx as nx

import heapq

from matplotlib.animation import FuncAnimation

class SensorNode:

def \_\_init\_\_(self, node\_id, energy, data, position):

self.node\_id = node\_id

self.energy = energy

self.data = data

self.position = position

def is\_alive(self):

return self.energy > 0

def harvest\_energy(self):

harvested\_energy = random.uniform(0, 5)

self.energy += harvested\_energy

print(f"Node {self.node\_id} harvested {harvested\_energy:.2f} energy.")

def collect\_data(self):

if self.is\_alive():

collected\_data = random.uniform(0, 10)

self.data += collected\_data

print(f"Node {self.node\_id} collected {collected\_data:.2f} data.")

return collected\_data

return 0

def encrypt\_data(self, data):

encrypted\_data = data \* 1.1 # Simulated encryption

print(f"Node {self.node\_id} encrypted data: {encrypted\_data:.2f}.")

return encrypted\_data

def transmit\_data(self, data, path):

if self.is\_alive() and self.energy > 0:

energy\_consumed = len(path) + (data / 10) # Consume energy based on the length of the path and amount of data

if self.energy >= energy\_consumed:

self.energy -= energy\_consumed

print(f"Node {self.node\_id} transmitted {data:.2f} to {path[-1]} via path {path}.")

return True

else:

print(f"Node {self.node\_id} has insufficient energy to transmit.")

return False

class ProxyBackend:

def process\_data(self, data):

processed\_data = data \* 0.9 # Simulated data processing

print(f"Processed data: {processed\_data:.2f}.")

return processed\_data

class WBAN:

def \_\_init\_\_(self, nodes\_positions, base\_station\_position, node\_energies):

self.nodes = []

self.proxy = ProxyBackend()

self.transmission\_paths = []

self.energy\_history = {}

# Initialize nodes with user-defined positions and energy levels

for i, position in enumerate(nodes\_positions):

node = SensorNode(f'Node {i}', energy=node\_energies[i], data=0, position=position)

self.nodes.append(node)

self.energy\_history[node.node\_id] = [] # Initialize energy history for each node

# Create the base station

self.base\_station = SensorNode("Base Station", 100, 0, base\_station\_position)

def link\_cost(self, node1, node2):

"""Calculate the link cost based on the energy levels of the nodes."""

distance = ((node1.position[0] - node2.position[0]) \*\* 2 +

(node1.position[1] - node2.position[1]) \*\* 2) \*\* 0.5

# Prevent division by zero by returning a large cost if either node's energy is zero

if node1.energy == 0 or node2.energy == 0:

return float('inf')

return distance / min(node1.energy, node2.energy) # Example cost function

def dijkstra(self, start\_node\_id):

graph = {node.node\_id: [] for node in self.nodes}

graph[self.base\_station.node\_id] = []

for node in self.nodes:

for other\_node in self.nodes:

if node.node\_id != other\_node.node\_id:

cost = self.link\_cost(node, other\_node)

graph[node.node\_id].append((other\_node.node\_id, cost))

graph[other\_node.node\_id].append((node.node\_id, cost)) # Bidirectional edges

# Add the base station edges

cost\_to\_bs = self.link\_cost(node, self.base\_station)

graph[node.node\_id].append((self.base\_station.node\_id, cost\_to\_bs))

# Dijkstra's algorithm

queue = [(0, start\_node\_id)] # (cost, node\_id)

distances = {node.node\_id: float('inf') for node in self.nodes}

distances[start\_node\_id] = 0

distances[self.base\_station.node\_id] = float('inf') # Initialize base station distance

shortest\_paths = {node.node\_id: [] for node in self.nodes}

visited = set()

while queue:

current\_distance, current\_node = heapq.heappop(queue)

visited.add(current\_node)

for neighbor, weight in graph[current\_node]:

if neighbor in visited:

continue

distance = current\_distance + weight

if distance < distances[neighbor]:

distances[neighbor] = distance

shortest\_paths[neighbor] = shortest\_paths[current\_node] + [current\_node]

heapq.heappush(queue, (distance, neighbor))

return shortest\_paths

def simulate(self, rounds):

for r in range(rounds):

print(f"\nRound {r + 1}")

for node in self.nodes:

if node.is\_alive():

node.harvest\_energy() # Harvest energy periodically

data = node.collect\_data()

if data > 0:

processed\_data = self.proxy.process\_data(data)

encrypted\_data = node.encrypt\_data(processed\_data)

# Find shortest path to the base station

shortest\_paths = self.dijkstra(node.node\_id)

path\_to\_bs = shortest\_paths[self.base\_station.node\_id] + [self.base\_station.node\_id]

if node.transmit\_data(encrypted\_data, path\_to\_bs):

self.transmission\_paths.append((node.node\_id, path\_to\_bs)) # Store transmission path

else:

print(f"Node {node.node\_id} has no energy to collect data.")

# Simulate node failure based on energy level

if node.energy < 5: # Example threshold

print(f"Node {node.node\_id} has failed due to low energy.")

node.energy = 0 # Mark node as failed

else:

print(f"Node {node.node\_id} is dead.")

# Store the current energy level for each node

self.energy\_history[node.node\_id].append(node.energy)

# Move nodes randomly after each round

for node in self.nodes:

if node.is\_alive():

node.position = (random.uniform(0, 10), random.uniform(0, 10)) # Randomly reposition nodes

self.animate\_data\_transmission()

def animate\_data\_transmission(self):

fig, ax = plt.subplots(figsize=(10, 8))

ax.set\_xlim(0, 10)

ax.set\_ylim(0, 10)

ax.set\_title("Data Transmission Animation")

ax.set\_xlabel("X Position")

ax.set\_ylabel("Y Position")

# Plot nodes

for node in self.nodes:

color = 'red' if node.is\_alive() else 'gray' # Gray for dead nodes

ax.scatter(node.position[0], node.position[1], c=color, s=100)

ax.text(node.position[0], node.position[1], node.node\_id, fontsize=12, ha='right')

# Plot the base station

ax.scatter(self.base\_station.position[0], self.base\_station.position[1], c='blue', s=200, marker='x')

ax.text(self.base\_station.position[0], self.base\_station.position[1], "Base Station", fontsize=12, ha='right')

# Create an empty line for the animation

line, = ax.plot([], [], 'g-', linewidth=2)

def init():

line.set\_data([], [])

return line,

def update(frame):

if frame < len(self.transmission\_paths):

start, path = self.transmission\_paths[frame]

path\_positions = []

for node\_id in path:

if node\_id == "Base Station":

path\_positions.append(self.base\_station.position) # Get base station position

else:

path\_positions.append(self.nodes[int(node\_id.split()[1])].position) # Extract node position

x\_data, y\_data = zip(\*path\_positions)

line.set\_data(x\_data, y\_data)

return line,

ani = FuncAnimation(fig, update, frames=len(self.transmission\_paths), init\_func=init, blit=True, repeat=False)

plt.show()

def plot\_energy\_levels(self):

plt.figure(figsize=(12, 6))

for node\_id, energies in self.energy\_history.items():

plt.plot(energies, label=node\_id)

plt.title("Energy Levels of Nodes Over Time")

plt.xlabel("Rounds")

plt.ylabel("Energy Level")

plt.legend()

plt.grid()

plt.show()

def get\_node\_positions\_and\_energy(num\_nodes):

"""Function to allow user to place nodes and assign energy levels on a plot."""

node\_positions = []

node\_energies = []

def onclick(event):

if event.xdata is not None and event.ydata is not None:

node\_positions.append((event.xdata, event.ydata))

plt.scatter(event.xdata, event.ydata, c='red', s=100)

plt.draw()

def on\_key(event):

if event.key == 'enter':

plt.close()

fig, ax = plt.subplots()

ax.set\_title("Place Sensor Nodes. Press Enter when done.")

plt.xlim(0, 10)

plt.ylim(0, 10)

cid\_click = fig.canvas.mpl\_connect('button\_press\_event', onclick)

cid\_key = fig.canvas.mpl\_connect('key\_press\_event', on\_key)

plt.show()

# Assign random energy levels for nodes

for \_ in range(num\_nodes):

node\_energies.append(random.uniform(10, 100)) # Random energy between 10 and 100

return node\_positions, node\_energies

def main():

num\_nodes = int(input("Enter the number of sensor nodes: "))

base\_station\_position = (5, 5) # Fixed position for simplicity

nodes\_positions, node\_energies = get\_node\_positions\_and\_energy(num\_nodes)

wban = WBAN(nodes\_positions, base\_station\_position, node\_energies)

rounds = int(input("Enter the number of simulation rounds: "))

wban.simulate(rounds)

wban.plot\_energy\_levels()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Performance Comparison:**

This is based on the study of various research paper

<https://ieeexplore.ieee.org/search/searchresult.jsp?newsearch=true&queryText=WBAN%20energy>

<https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8276162>

<https://ieeexplore.ieee.org/document/9443125>

<https://ieeexplore.ieee.org/document/7811040>

| **Performance Metrics** | **Dijkstra-based Algorithm** | **AODV** | **DSR** | **Ehyaie et al. (2009)** | **Guo et al. (2010)** |
| --- | --- | --- | --- | --- | --- |
| Packet Delivery Ratio | High | Moderate | Moderate | Low | Moderate |
| End-to-End Delay | Low | High | High | Moderate | Moderate |
| Throughput | High | Moderate | Moderate | Low | Moderate |
| Energy Efficiency | High | Low | Low | Moderate | Moderate |

**Conclusion:**

The project optimizes WBAN energy efficiency using a modified Dijkstra algorithm, focusing on balanced energy use across nodes. It achieves high packet delivery, low delay, and extends network life. Performance surpasses AODV and DSR in energy efficiency and throughput