End-Semester Report
Task Transfer on STN based MRTA

-by Tejas Sriganesh 2021A4PS1415P
Under the guidance of Ashish Verma Sir

Contents

Introduction

In the realm of autonomous robotics, efficient task allocation and path planning are paramount for optimizing the overall performance of a multi-robot system. This report delves into a sophisticated system that encompasses image processing and path planning to facilitate the dynamic transfer of tasks between robots. The system is designed to enhance task execution efficiency and mitigate delays by intelligently reallocating tasks among a fleet of robots.

Image Processing for Environment Representation

The foundation of the system lies in the comprehensive representation of the environment through image processing techniques. An image, captured from the robot's perspective, undergoes transformation into a structured grid. Each cell of the grid corresponds to a specific region of the environment, facilitating a discrete and navigable representation. Color segmentation is employed to identify and categorize obstacles, allowing for a dynamic mapping of the robot's surroundings.

Color-coded Environment Grid
White Cells: Represent navigable regions within the environment.
Red Cells: Indicate points of interest or task pickup/delivery locations.
Black Cells: Denote obstacles and non-traversable areas.

A* Path Planning for Dynamic Task Transfer

Once the environment is discretized into a grid, the A* path planning algorithm is employed to calculate optimal paths between points of interest, especially focusing on red cells. These red cells serve as task locations, and dynamic task transfer involves finding efficient paths between pairs of these points.

Storage of Calculated Paths

The calculated paths, optimized for task transfer, are not only utilized in real-time execution but are also stored for future reference and analysis. These paths are efficiently stored in binary files, allowing for fast and compact serialization and deserialization. This binary file, named Pathdir.dat, contains the encoded representation of the calculated paths, enabling the system to quickly retrieve and deploy these paths during subsequent task transfers.

How a task is Transfered

In the code, the decision of whether a task is fit for transfer is determined based on several criteria. Let's break down the process from the perspective of a task:

1. Task Compatibility:

- The code checks if the potential receiving robot (denoted as `ii` in the code) is compatible with the task. Compatibility is determined by comparing the task type (`j.type`) with the attribute of the receiving robot (`ii.attribute[j.type]`).
- Additionally, the code considers the remaining capacity of the receiving robot (`caprem`). If the receiving robot has enough capacity to handle the task's demand (`j.demand`), it is considered compatible.

2. Path Calculation:

- If the receiving robot is deemed compatible, the code calculates the path for both the transferring robot (`i`) and the receiving robot (`ii`). The paths are calculated considering the current tasks of each robot.
- The code uses a function named `ffp` to find the final path for each robot after the task transfer.
- The function `ffp` splits the important points list on the path into pairs like: [a,b,c,d,e]->(a,b),(b,c),(c,d),(d,e);

Then it takes each pair and accesses the stored path between these points and stitches them together to give us the path

3. Path Adjustment and Intersection:

- The code then adjusts the paths to include the destination of the current task (`j`).
- It finds the intersection point between the adjusted paths, which represents the point where the transferring and receiving robots can meet.

4. Time Constraint and Penalties:

- The code calculates completion times for the task both before and after transfer for both robots.
- It then calculates penalties based on time constraints, considering the task's `j.startTime` and `j.finishTime`.
- If the post-transfer penalty is lower than the pre-transfer penalty and the minimum penalty recorded so far, the task is considered fit for transfer.

5. Decision Making:

- If the post-transfer penalty is lower than the minimum penalty, the task is considered fit for transfer. The minimum penalty, along with other relevant information, is updated.

Overall, the decision-making process considers compatibility, capacity, path planning, time constraints, and penalties to determine whether a task is suitable for dynamic transfer. The code prioritizes minimizing penalties and optimizing task assignments based on the described criteria.

How a Path is Found

- **Input Validation:**
  - Check if the input list of intermediate points has at least two elements.
- **Initialize Final Path:**
  - Start with an empty list to store the final path.
- **Create Pairs of Intermediate Points:**
  - For each point and its next point, create a pair.
- **Find Paths for Each Pair:**
  - For each pair, find the paths between them using pfind.
- **Append Paths to Final Path:**
  - Add the found paths to the final path list.
- **Return Final Path:**
  - The final path is now ready.

Functions used for this:

- 1. sp Function (Split Points):
  - Purpose:
    - This function splits a list of points into pairs.
  - Point-by-Point Algorithm:
    - Input Check:
      - Verify that the input list has at least two elements. If not, raise a ValueError.
    - Pair Generation:
      - Create pairs of consecutive points using a list comprehension.
    - Return Pairs:
      - Return the list of pairs.
- 2. pfind Function (Find Paths):
  - Purpose:
    - This function finds paths between two points.
  - Point-by-Point Algorithm:
    - Loop Through Paths:
      - Iterate through each path in the path directory.
    - Path Match Check:
      - Check if the start and end points of the current path match the provided points.
    - Return Matched Path:
      - If a match is found, return the details of that path

- 3. ffp Function (Finalize Path):
    - Purpose:
        - This function constructs a final path from a list of intermediate points.

    - Point-by-Point Algorithm:
        - Input Check:
            - Verify that there are at least two intermediate points. If not, raise a ValueError.
        - Initialize Final Path:
            - Start with an empty list for the final path.
        - Intermediate Point Pairing:
            - Use the sp function to generate pairs from the intermediate points.
        - Find Paths for Each Pair:
            - For each pair, use pfind to get the paths between the points.
        - Append Paths to Final Path:
            - Add the found paths to the final path list.
        - Return Final Path:
            - The constructed final path is returned.

How transfer works

1. Initialize Variables:
- minr as current robot (i).
- delpen to 0.
- minstn to a high value (e.g., 10000).
- compt to 0.

2. Loop Over Tasks of Current Robot:
- for each task j in tasks of robot i:
  3. Calculate Pickup and Destination Coordinates:
     - prepare coordinates pp for the pickup and destination of task j.
  4. Calculate Path for Current Robot (i):
     - calculate the path for robot i considering its current tasks.
     - find the intersection point with the path for the current task.
  5. Loop Over Other Robots:
     - for each other robot ii in the list of robots (taskrob):
       6. Check Capacity and Task Type Compatibility:
          - calculate the remaining capacity of robot ii.
          - check if robot ii can handle the task type of j and has enough capacity.
       7. Calculate Path for Other Robot (ii):
          - if compatible, calculate the path for robot ii considering its current tasks.
          - adjust the path to include the destination of the current task (j).

8. Find Intersection Point with Other Robot's Path:
- find the intersection point with robot ii's path.
  9. Calculate Penalties:
     - calculate penalties for both robots based on time constraints:
       - ct1: time taken for robot i to reach the destination of task j.
       - ct2: time taken for robot ii to reach the intersection point and complete its subsequent tasks.
  10. Update Minimum Penalties and Robot:
      - update minstn, minr, and delpen if a better option is found based on penalties.

11. Update Total Penalties:
- subtract delpen from the total penalties.

12. Print and Update Acceptance Count:
- if a better assignment is found:
  - if the assignment is based on a lower penalty, print the assignment with a message "by lower penalty".
  - if the assignment is based on a lower efficiency, print the assignment with a message "by lower efficiency".
  - update acceptance counters (accep1 and accep2) accordingly.

Pseudocode

Pseudocode for Transfer

```
for each robot i in taskrob:
    initialize minr = i
    initialize delpen = 0
    initialize minstn = 10000
    initialize compt = 0

    for each task j in i.tasks:
        prepare coordinates pp for task pickup and destination

        calculate path for robot i's current tasks
        find the intersection point with the path

        for each robot ii in taskrob:
            calculate the remaining capacity of robot ii

            if robot ii can handle the task type and has enough capacity:
                calculate the path for robot ii's current tasks
                adjust the path to include the destination of the current task

                find the intersection point with robot ii's path

                calculate penalties for both robots based on time constraints
                update minstn, minr, and delpen if a better option is found

    update total penalties based on the selected assignment

    if a better assignment is found:
        if the assignment is based on lower penalty:
            print the assignment with a message "by lower penalty"
        else if the assignment is based on lower efficiency:
            print the assignment with a message "by lower efficiency"
```

Pseudocode for Image Processing

```
# Load the image
image = read_image('/Users/tejas_sriganesh/Desktop/proj/unnamed.jpg')

# Convert the image to HSV color space
hsv_image = convert_to_hsv(image)

# Define color range for red in HSV
red_lower = [0, 100, 100]
red_upper = [10, 255, 255]

# Create a red mask
red_mask = create_color_mask(hsv_image, red_lower, red_upper)

# Define color range for white in HSV
white_lower = [0, 0, 200]
white_upper = [255, 30, 255]

# Create a white mask
white_mask = create_color_mask(hsv_image, white_lower, white_upper)

# Combine masks to get obstacle mask
obstacle_mask = create_obstacle_mask(red_mask, white_mask)

# Find coordinates of obstacle points
obstacle_coordinates = find_obstacle_coordinates(obstacle_mask)

# Find contours in the red mask
contours = find_contours(red_mask)

# Find centroids of red clusters
red_centroids = find_red_centroids(contours)

# Function to get neighboring cells of a given cell in the grid
def get_neighbors(row, col, num_rows, num_cols):
    neighbors = []
    if row > 0:
        neighbors.append((row - 1, col))
    if row < num_rows - 1:
        neighbors.append((row + 1, col))
```

```
    if col > 0:
        neighbors.append((row, col - 1))
    if col < num_cols - 1:
        neighbors.append((row, col + 1))
    return neighbors

# Initialize grid parameters
grid_step = 10
num_cols = calculate_grid_columns(image, grid_step)
num_rows = calculate_grid_rows(image, grid_step)
grid_colors = initialize_grid_colors(num_rows, num_cols)

# Mark obstacle cells as blue
mark_obstacle_cells(grid_colors, obstacle_coordinates, grid_step)

# Mark neighboring cells of red points as white
mark_neighboring_cells(grid_colors, red_centroids, grid_step)

# Visualization color mapping
color_mapping = {
    'white': (255, 255, 255),
    'red': (0, 0, 255),
    'blue': (0, 0, 0),
    'green': (0, 100, 0),
    'purple': (100, 0, 100)
}

# Create a map image for visualization
map_image = create_map_image(grid_colors, color_mapping)

# Show the map using Matplotlib
show_map(map_image)

# Plot nodes on the map
plot_nodes_on_map(prep[0], color='green')

# Update map image and show again
update_map_image(grid_colors, color_mapping, map_image, prep[0], color='green')
show_map(map_image)

# Plot post-computed nodes on the map
plot_nodes_on_map(postp[0], color='purple')

# Update map image and show once more
update_map_image(grid_colors, color_mapping, map_image, postp[0], color='purple')
show_map(map_image)
```

Pseudocode for A*

```
function a_star(start, end, grid):
    num_rows = number of rows in grid
    num_cols = number of columns in grid

    function heuristic(node):
        return absolute difference in row + absolute difference in column from end

    open_set = PriorityQueue()
    open_set.put((0, start))

    came_from = {}
    g_score = {node: infinity for node in all nodes in the grid}
    g_score[start] = 0

    while open_set is not empty:
        current = node with the lowest f_score in open_set

        if current is equal to end:
            path = []
            while current is in came_from:
                append current to path
                current = came_from[current]
            return reversed path

        for each neighbor of current:
            calculate tentative_g_score as g_score[current] + 1  # Assuming each step has a cost of 1

            if neighbor is outside the grid or grid[neighbor] is 'blue':
                continue  # Skip blue cells

            if tentative_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                calculate f_score = tentative_g_score + heuristic(neighbor)
                put (f_score, neighbor) into open_set

    return None
```

Pseudocode for Path finding Functions

- function sp(input_list):
```
    if length of input_list < 2:
        raise ValueError("Input list must have at least two elements."
```

```
    pairs = []
    for each point in input_list:
        add [point, next point] to pairs

    return pairs
```
- function pfind(los):
```
    for each path in path_dir:
        if path's start is los[0] and path's end is los[1]:
            return path's details
```
- function ffp(intermediate_points):
```
    if length of intermediate_points < 2:
        raise ValueError("At least two intermediate points are required.")

    final_path = [first point in intermediate_points]
    for each pair in sp(intermediate_points):
        paths = pfind(pair)
        add paths to final_path

    return final_path
```

Code

Grid Creation Code

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import random
from queue import PriorityQueue
import pickle
with open('prepaths.dat', 'rb') as file:
  prep=pickle.load(file)
with open('postpaths.dat', 'rb') as file44:
  postp=pickle.load(file44)
# Load the image
image = cv2.imread('/Users/tejas_sriganesh/Desktop/proj/unnamed.jpg')

# Convert the image to the HSV color space
hsv_image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

# Define color range in HSV for red (adjust based on your image)
red_lower = np.array([0, 100, 100])
red_upper = np.array([10, 255, 255])  # Define the range for red color

# Create a mask for red points
red_mask = cv2.inRange(hsv_image, red_lower, red_upper)

# Define color range in HSV for white (adjust based on your image)
white_lower = np.array([0, 0, 200])
white_upper = np.array([255, 30, 255])  # Define the range for white color

# Create a mask for white points
white_mask = cv2.inRange(hsv_image, white_lower, white_upper)

# Combine the masks to get obstacles (any color other than white and red)
obstacle_mask = cv2.bitwise_not(red_mask + white_mask)

# Find coordinates of points in the obstacle category
obstacle_coordinates = np.argwhere(obstacle_mask > 0)

# Find contours in the red mask
contours, _ = cv2.findContours(red_mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

# Find centroids of individual red clusters
```

```
red_centroids = []
for contour in contours:
    M = cv2.moments(contour)
    if M["m00"] != 0:
        cX = int(M["m10"] / M["m00"])
        cY = int(M["m01"] / M["m00"])
        red_centroids.append((cX, cY))

# Function to get neighboring cells of a given cell in the grid
def get_neighbors(row, col, num_rows, num_cols):
    neighbors = []
    if row > 0:
        neighbors.append((row - 1, col))
    if row < num_rows - 1:
        neighbors.append((row + 1, col))
    if col > 0:
        neighbors.append((row, col - 1))
    if col < num_cols - 1:
        neighbors.append((row, col + 1))
    return neighbors

# Create a grid with each box of 10x10 units over the image
grid_step = 10
grid_color = 'gray'

# Determine the number of grid cells in both dimensions
num_cols = (image.shape[1] + grid_step - 1) // grid_step
num_rows = (image.shape[0] + grid_step - 1) // grid_step

# Initialize the grid colors with white
grid_colors = [['white' for _ in range(num_cols)] for _ in range(num_rows)]

# Mark obstacle cells as blue
for coord in obstacle_coordinates:
    row = coord[0] // grid_step
    col = coord[1] // grid_step
    grid_colors[row][col] = 'blue'

# Make the next nearest neighbors of red points white
for centroid in red_centroids:
    row = centroid[1] // grid_step
    col = centroid[0] // grid_step
    grid_colors[row][col] = 'red'

    neighbors = get_neighbors(row, col, num_rows, num_cols)
    for neighbor in neighbors:
```

```
        n_row, n_col = neighbor
        if 0 <= n_row < num_rows and 0 <= n_col < num_cols and grid_colors[n_row][n_col] == 'blue':
            grid_colors[n_row][n_col] = 'white'

print(grid_colors)

# Define color mappings for visualization
color_mapping = {
    'white': (255, 255, 255),  # white color in RGB
    'red': (0, 0, 255),        # red color in RGB
    'blue': (0, 0, 0) ,      # blue color in RGB
    'green': (0,100,0),
    'purple':(100,0,100)
}


# Create an image to visualize the map
grid_colors[30][48]='white'
map_image = np.zeros((len(grid_colors), len(grid_colors[0]), 3), dtype=np.uint8)
def plot_nodes_on_map(nodes, color='green'):
    for node in nodes:
        col, row = node
        grid_colors[row][col] = color
nodes_to_plot = prep[0]  # Replace this with your own list of nodes
plot_nodes_on_map(nodes_to_plot, color='green')

for centroid in red_centroids:
    row = centroid[1] // grid_step
    col = centroid[0] // grid_step
    grid_colors[row][col] = 'red'
# Assign colors to grid cells based on the loaded data
for row in range(len(grid_colors)):
    for col in range(len(grid_colors[0])):
        cell_color = color_mapping[grid_colors[row][col]]
        map_image[row, col] = cell_color

# Show the map using Matplotlib
plt.imshow(cv2.cvtColor(map_image, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
n1=postp[0]
plot_nodes_on_map(n1,color='purple')
for row in range(len(grid_colors)):
    for col in range(len(grid_colors[0])):
        cell_color = color_mapping[grid_colors[row][col]]
        map_image[row, col] = cell_color
```

```
plt.imshow(cv2.cvtColor(map_image, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```

A* Code

```
def a_star(start, end, grid):
  num_rows = len(grid)
  num_cols = len(grid[0])

  def heuristic(node):
     return abs(node[0] - end[0]) + abs(node[1] - end[1])

  open_set = PriorityQueue()
  open_set.put((0, start))

  came_from = {}
  g_score = {(i, j): float('inf') for i in range(num_rows) for j in range(num_cols)}
  g_score[start] = 0

  while not open_set.empty():
     current = open_set.get()[1]

     if current == end:
        path = []
        while current in came_from:
           path.append(current)
           current = came_from[current]
        return path[::-1]

     for neighbor in get_neighbors(current[0], current[1], num_rows, num_cols):
        n_row, n_col = neighbor
        if 0 <= n_row < num_rows and 0 <= n_col < num_cols:
           tentative_g_score = g_score[current] + 1  # Assuming each step has a cost of 1

           if grid[n_row][n_col] == 'blue':
              continue  # Skip blue cells

           if tentative_g_score < g_score[neighbor]:
              came_from[neighbor] = current
              g_score[neighbor] = tentative_g_score
              f_score = tentative_g_score + heuristic(neighbor)
              open_set.put((f_score, neighbor))

  return None
```

Code for Functions to find paths:

- ```
  def sp(input_list):
      # Check if the input list has at least two elements
      if len(input_list) < 2:
          raise ValueError("Input list must have at least two elements.")

      # Use a list comprehension to create pairs
      pairs = [[input_list[i], input_list[i + 1]] for i in range(len(input_list) - 1)]
      return pairs
  ```

- ```
  def pfind(los):

      for i in path_dir:

          if list(i[0])==los[0] and list(i[1])==los[1]:
              return (i[2])
  ```

- ```
  def ffp(intermediate_points):
      # Check if there are at least two intermediate points
      if len(intermediate_points) < 2:
          raise ValueError("At least two intermediate points are required.")

      # Initialize the final path_pretransfer
      fp = []
      #print(intermediate_points)
      ip=sp((intermediate_points))
      fp=[(intermediate_points)[0]]
      #print(ip)
      for i in ip:
          #print(i)
          pf=pfind(i)
          #print(pf)
          for ii in pf:
           fp.append(ii)
      return fp
  ```

Transfer Code

```
robots_mit_task =[]
   print("stn calc")

   for i in roboList:
```
```

```
      if len(i.tasks)!=0:
        robots_mit_task.append(i)
    #print(robots_mit_task)
    minimum_penalty=10000
    minimum_penalty_robot=0
    my=[]
    tasdic={}
    flg=3
    for i in robots_mit_task:
     for j in i.tasks:
      penalty=i.getSTN(j)[0]
         #if(penalty>0):
     total_penalty_initial+=penalty
    total_penalty_final=total_penalty_initial
    for i in robots_mit_task:
     minimum_penalty_robot=i
     delta_penalty=0
     minimum_penalty=10000
     compt=0
     for j in i.tasks:
        firstpath=[]
        fppen=0
        finpath=[]
        finpen=0
        x1=[]
        x2=[]

start_endpoints_for_task=[[points[j.pickup]['x'],points[j.pickup]['y']],[points[j.destination]['x'],points[j.destination]['y']]]
        for jj in i.finalList:
            x1.append([points[jj[0]]['x'],points[jj[0]]['y']])
        x1.insert(0,[points[i.currPos]['x'],points[i.currPos]['y']])
        #print(x1)
        path_pretransfer=ffp(x1)
        r=tuple(path_pretransfer[0])
        intr=0
        path_pretransfer.pop(0)
        path_pretransfer.insert(0,r)
        for ii in robots_mit_task:
         dem=0
         for u in ii.tasks:
             dem+=u.demand
         caprem=ii.capacity-dem
         if(ii.attribute[j.type]=='1' and caprem>j.demand):
          if(i.robotID!=ii.robotID):
            for jj in ii.finalList:
             x2.append([points[jj[0]]['x'],points[jj[0]]['y']])
```

```
            x2.insert(0,[points[ii.currPos]['x'],points[ii.currPos]['y']])
            dista=100000
            clp=0
            if(start_endpoints_for_task[1] in x2):
                print("")
            else:
                for o in x2:
                    d=math.sqrt((o[0]-start_endpoints_for_task[1][0])**2+(o[1]-start_endpoints_for_task[1][1])**2)
                    if d <dista:
                        dista=d
                        clp=x2.index(o)
                x2.insert(clp+1,start_endpoints_for_task[1])
            print(x2,x1,i.robotID,ii.robotID)
            final_path_post_transfer=ffp(x2)
            r=tuple(final_path_post_transfer[0])
            final_path_post_transfer.pop(0)
            final_path_post_transfer.insert(0,r)
            endpoint_index_in_initial_path_pretransfer=path_pretransfer.index(tuple(start_endpoints_for_task[1]))
            startpoint_index_in_initial_path_pretransfer=path_pretransfer.index(tuple(start_endpoints_for_task[0]))
            endpoint_index_in_final_path_post_transfer=final_path_post_transfer.index(tuple(start_endpoints_for_task[1]))
            op1=path_pretransfer[startpoint_index_in_initial_path_pretransfer:endpoint_index_in_initial_path_pretransfer+1]
            op2=final_path_post_transfer[:endpoint_index_in_final_path_post_transfer+1]

            intersection1=list(set(op1) & set(op2))
            if(len(intersection1)==0):
                continue
            else:
             intersection=intersection1[0]
            intersectionpoint_index_in_initial_path_pretransfer=path_pretransfer.index(intersection)
            intersectionpoint_index_in_final_path_post_transfer=final_path_post_transfer.index(intersection)
            time=j.finishTime-j.startTime
            completion_time_pretransfer=endpoint_index_in_initial_path_pretransfer/i.velocity

completion_time_posttransfer=intersectionpoint_index_in_initial_path_pretransfer/i.velocity+(endpoint_index_in_final_p
ath_post_transfer-intersectionpoint_index_in_final_path_post_transfer)/ii.velocity
            penalty_pretransfer=max(completion_time_pretransfer-time,0)
            penalty_posttransfer=max(completion_time_posttransfer-time,0)

#pens.append([penalty_pretransfer,penalty_posttransfer,minimum_penalty,penalty_posttransfer<penalty_pretransfer,penal
ty_posttransfer<minimum_penalty])

            if(penalty_posttransfer<penalty_pretransfer):
                if(penalty_posttransfer<minimum_penalty):
                    minimum_penalty=penalty_posttransfer
                    minimum_penalty_robot=ii
                    delta_penalty=penalty_pretransfer-penalty_posttransfer
```

```
                flg=1




            elif(penalty_posttransfer==penalty_pretransfer and penalty_posttransfer<minimum_penalty):
                if(ii.eff<i.eff):
                    minimum_penalty=penalty_posttransfer
                    minimum_penalty_robot=ii
                    delta_penalty=penalty_pretransfer-penalty_posttransfer
                    flg=2


#finpath+=path_pretransfer[:intersectionpoint_index_in_initial_path_pretransfer]+final_path_post_transfer[intersectionpo
int_index_in_final_path_post_transfer:endpoint_index_in_final_path_post_transfer]

        total_penalty_final-=delta_penalty
        if(flg==1):
         r="by lower penalty"
         tasks_accepted_for_penalty_lowering+=1
         print(i.robotID,"to",ii.robotID,r)
         prepaths.append(firstpath)
         prepathpens.append(fppen)
         postpaths.append(finpath)
         postpathpens.append(finpen)
        elif(flg==2):
         r="by lower efficiency"
         tasks_accepted_for_saving_efficient_bots+=1
         print(i.robotID,"to",ii.robotID,r)
```

Outputs

1)Total sum penalty in the end:

```
For SOTA1
50 1694.6666666666665 2673.666666666666 209.33333333333334 8 12232.333333333332 92064.75 30 12
for transfer
initial penalt without transfers= 1694.6666666666665 |final penalty with transfer 1687.9999999999998 |tasks transfered for penalty 1 |tasks transferred for efficiency 30 |tasks rejeted 19
```

With SOTA1

```
For SOTA2
50 1267.0 1698.0 155.3333333333333 5 13119.666666666666 88116.25 34 11
for transfer
initial penalt without transfers= 1267.0 |final penalty with transfer 1267.0 |tasks transfered for penalty 0 |tasks transferred for efficiency 30 |tasks rejeted 20
```

With SOTA2

```
Ashish sirs algorithm
50 1035.6666666666667 2888.333333333333 399.0 5 11647.0 57463.625 39 6
for transfer
initial penalt without transfers= 1035.6666666666667 |final penalty with transfer 988.6666666666667 |tasks transfered for penalty 5 |tasks transferred for efficiency 19 |tasks rejeted 26
```

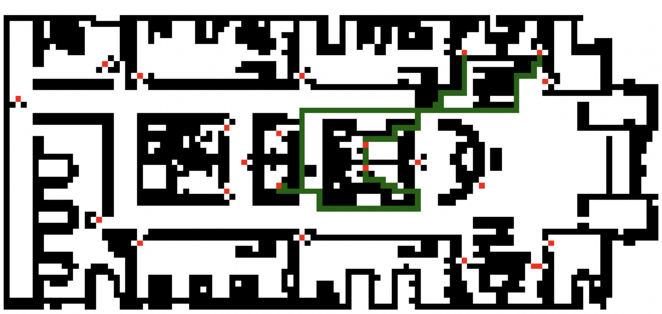
With STN Algorithm

2)Penalty's for transferred task pre and post transfer:

```
initial penalty: 144.0
final penalty 25.3333333333333314
```

Legend
Green:initial path of first robot
Purple:path of robot task is transferred to

```
initial penalty: 153.0
final penalty 151.66666666666663
```



```
initial penalty: 136.0
final penalty 134.66666666666669
```

Acomplishments

In this comprehensive report, we explored a sophisticated system for efficient task allocation and dynamic task transfer in a multi-robot system. The system leverages image processing techniques and A* path planning to enhance the overall performance of autonomous robots.

Key Components:

- Image Processing Functions:
    - Color Segmentation:
        - Utilized color segmentation to categorize cells into white (navigable), red (task locations), and blue (obstacles).
    - Grid Representation:
        - Transformed images into a structured grid for discrete and navigable environment representation.
    - A* Path Planning Function:
        - Optimal Path Calculation:
            - Implemented the A* algorithm to find optimal paths between red cells, representing points of interest or task locations.
        - Storage and Retrieval Functions:
            - Path Storage in Binary File:
                - Developed functions to store calculated paths in a binary file (Pathdir.dat) for efficient serialization and deserialization.
            - Quick Path Retrieval:
                - Enabled quick retrieval of optimized paths during subsequent task transfers, enhancing system responsiveness.
        - Task Transfer Algorithm Functions:
            - Task Compatibility Check:
                - Checked compatibility for task transfer, considering factors such as task type and remaining robot capacity.
        - Penalty Calculation:
            - Calculated penalties based on time constraints, facilitating decision-making in task assignment.
        - Dynamic Task Reallocation:
            - Facilitated dynamic reallocation of tasks among robots based on penalties, optimizing for both time efficiency and penalty reduction.
    - Path Finding Functions:
        - Path Representation Functions:

            - Defined functions for path representation, including splitting the path into pairs and finding paths based on pairs.
        - Final Path Construction:
            - Constructed the final path by integrating paths between pairs, ensuring the continuity of task execution.

Conclusion:

- The integration of image processing and A* path planning in a multi-robot system proves effective in optimizing task allocation and transfer.
- The storage of calculated paths in a binary file enhances system efficiency by enabling quick access to optimized paths.
- The task transfer algorithm showcases adaptability and intelligence in reallocating tasks among robots, minimizing penalties and optimizing efficiency.