Data Science Assignment 4

Nike Marie Pulow – Henri Paul Heyden
stu239549 – stu240825

1 PCA, SVD

1. We search for matrices U, Σ, V with $X = U\Sigma V^T$ so that the columns of U and V are eigenvectors of XX^T and $X^T X$ respectively.
We'll begin by computing XX^T:

$$\begin{aligned} XX^T &= \begin{bmatrix} 4 & 2 \\ \sqrt{2} & 2\sqrt{2} \\ -\sqrt{2} & -2\sqrt{2} \end{bmatrix} \times \begin{bmatrix} 4 & \sqrt{2} & -\sqrt{2} \\ 2 & 2\sqrt{2} & -2\sqrt{2} \end{bmatrix} \\ &= \begin{bmatrix} 20 & 8\sqrt{2} & -8\sqrt{2} \\ 8\sqrt{2} & 10 & -10 \\ -8\sqrt{2} & -10 & 10 \end{bmatrix} \end{aligned}$$

Now we have to solve $\det(X - \lambda I_{3\times3}) = 0$ for the eigenvalues of XX^T.
An expanded form of this equation is:

$$\begin{aligned} 0 &= \det(X - \lambda I_{3\times3}) \\ &= \det\left(\begin{bmatrix} 20 & 8\sqrt{2} & -8\sqrt{2} \\ 8\sqrt{2} & 10 & -10 \\ -8\sqrt{2} & -10 & 10 \end{bmatrix} - \begin{bmatrix} \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & \lambda \end{bmatrix} \right) \\ &= \det\left(\begin{bmatrix} 20-\lambda & 8\sqrt{2} & -8\sqrt{2} \\ 8\sqrt{2} & 10-\lambda & -10 \\ -8\sqrt{2} & -10 & 10-\lambda \end{bmatrix} \right) \\ &= (20-\lambda)(10-\lambda)(10-\lambda) + (8\sqrt{2})(-10)(-8\sqrt{2}) + (-8\sqrt{2})(8\sqrt{2})(-10) \\ &\quad -(-8\sqrt{2})(10-\lambda)(-8\sqrt{2}) - (8\sqrt{2})(8\sqrt{2})(10-\lambda) - (20-\lambda)(-10)(-10) \end{aligned}$$

This is a third degree polynomial equation with expanded form $-x^3 + 40x^2 - 144x = 0$ [1]
We get the solutions:

$$\lambda_1 = 36 \wedge \lambda_2 = 4 \wedge \lambda_3 = 0$$

With this we have

$$\Sigma = \begin{bmatrix} 6 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

With the eigenvalues of XX^T being calculated, we can now solve for the eigenvectors of XX^T:

$$\begin{aligned} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} &= \left(\begin{bmatrix} 20 & 8\sqrt{2} & -8\sqrt{2} \\ 8\sqrt{2} & 10 & -10 \\ -8\sqrt{2} & -10 & 10 \end{bmatrix} - \begin{bmatrix} 36 & 0 & 0 \\ 0 & 36 & 0 \\ 0 & 0 & 36 \end{bmatrix} \right) \times \begin{bmatrix} v_{1_1} \\ v_{1_2} \\ v_{1_3} \end{bmatrix} \\ &= \begin{bmatrix} -16 & 8\sqrt{2} & -8\sqrt{2} \\ 8\sqrt{2} & -26 & -10 \\ -8\sqrt{2} & -10 & -26 \end{bmatrix} \times \begin{bmatrix} v_{1_1} \\ v_{1_2} \\ v_{1_3} \end{bmatrix} \\ &= \begin{bmatrix} -16v_{1_1} + 8\sqrt{2}v_{1_2} - 8\sqrt{2}v_{1_3} \\ 8\sqrt{2}v_{1_1} - 26v_{1_2} - 10v_{1_3} \\ -8\sqrt{2}v_{1_1} - 10v_{1_2} - 26v_{1_3} \end{bmatrix} \end{aligned}$$

[1] Actually writing each step of this down would take too much space.

When numbering the equations in collumns **I**, **II** and **III** we can see that $\mathbf{I} \Rightarrow v_{1_1} = \frac{\sqrt{2}v_{1_2}}{2} - \frac{\sqrt{2}v_{1_3}}{2}$ and $\mathbf{II} + \mathbf{III} \Rightarrow v_{1_2} = -v_{1_3}$ which results in:

$$v_{1_1} = \sqrt{2}v_{1_2}$$

and respectively:

$$v_{1_2} = \frac{v_{1_1}}{\sqrt{2}} \wedge v_{1_3} = -\frac{v_{1_1}}{\sqrt{2}}$$

There are infinite solutions for this, with one being:

$$v_{1_1} = 1 \wedge v_{1_2} = \sqrt{2}^{-1} \wedge v_{1_3} = -\sqrt{2}^{-1}$$

So we get one eigenvector:

$$v_1 = \begin{bmatrix} 1 \\ \sqrt{2}^{-1} \\ -\sqrt{2}^{-1} \end{bmatrix}$$

Because writing the processes down takes a lot of space, the following linear equation systems will not be solved, but solutions provided. We have already demonstrated that we can solve one and the probability of minor mistakes when writing it down ruining the whole task is too great.
The second eigenvector we get by this time subtracting the diagonal matrix times 4 instead of 36 and we get a solution:

$$v_2 = \begin{bmatrix} 1 \\ -\sqrt{2}^{-1} \\ \sqrt{2}^{-1} \end{bmatrix}$$

And the third eigenvector is quite trivial:

$$v_3 = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$$

Something that wasn't specified in the slides was that the eigenvectors must be normalized so that U and V are orthonormal.
For that we have to divide all these vectors by $\sqrt{2}$.
We get:

$$U = \begin{bmatrix} \sqrt{2}^{-1} & \sqrt{2}^{-1} & 0 \\ 1/2 & -1/2 & \sqrt{2}^{-1} \\ -1/2 & 1/2 & \sqrt{2}^{-1} \end{bmatrix}$$

For a quick verification, we can compute UU^T and yes we get $I_{3\times3}$, so U is indeed orthonormal.
Now to V
As the lecture specifies, the eigenvalues of XX^T are the same like those of X^TX.
This means we „just" have to solve the linear equations but for X^TX and not XX^T.
We'll begin by computing X^TX:

$$\begin{aligned} X^TX &= \begin{bmatrix} 4 & \sqrt{2} & -\sqrt{2} \\ 2 & 2\sqrt{2} & -2\sqrt{2} \end{bmatrix} \times \begin{bmatrix} 4 & 2 \\ \sqrt{2} & 2\sqrt{2} \\ -\sqrt{2} & -2\sqrt{2} \end{bmatrix} \\ &= \begin{bmatrix} 20 & 16 \\ 16 & 20 \end{bmatrix} \end{aligned}$$

Thank god that this time it's small and regular.
We get the following systems of linear equations: With eigenvalue 36 we get:

$$\begin{aligned} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} &= \begin{bmatrix} -16 & 16 \\ 16 & -16 \end{bmatrix} \times \begin{bmatrix} v_{1_1} \\ v_{1_2} \end{bmatrix} \\ &= \begin{bmatrix} 16v_{1_1} - 16v_{1_2} \\ 16v_{1_1} - 16v_{1_2} \end{bmatrix} \end{aligned}$$

With one solution being:

$$v_1 = \begin{bmatrix} \sqrt{2}^{-1} \\ \sqrt{2}^{-1} \end{bmatrix}$$

With eigenvalue 4 we get:

$$\begin{aligned} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} &= \begin{bmatrix} 16 & 16 \\ 16 & 16 \end{bmatrix} \times \begin{bmatrix} v_{2_1} \\ v_{2_2} \end{bmatrix} \\ &= \begin{bmatrix} 16v_{2_1} + 16v_{2_2} \\ 16v_{2_1} + 16v_{2_2} \end{bmatrix} \end{aligned}$$

With one solution being

$$v_2 = \begin{bmatrix} \sqrt{2}^{-1} \\ -\sqrt{2}^{-1} \end{bmatrix}$$

For the last eigenvalue 0 we get $v_3 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, which formally isn't an eigenvector, since it would be an eigenvector of every matrix, but it is the only solution to the corresponding eigenvalue.
With that we get:

$$V^T = \begin{bmatrix} \sqrt{2}^{-1} & \sqrt{2}^{-1} \\ \sqrt{2}^{-1} & -\sqrt{2}^{-1} \\ 0 & 0 \end{bmatrix}$$

And now we can calculate the product $U\Sigma V^T$ to verify that it is actually right, which I did, but to be honest, writing all that down with each addition and so on would take a lot of time. ☐

2. We firstly have to center the points, since to calculate the eigenvectors of the covariance Matrix, we can't assume that the mean x and y components are 0. We get:

$$\overline{x} = 4/3 \wedge \overline{y} = 2/3$$

With this we can center X:

$$X_C = \begin{bmatrix} 4 - 4/3 & 2 - 2/3 \\ \sqrt{2} - 4/3 & 2\sqrt{2} - 2/3 \\ -\sqrt{2} - 4/3 & -2\sqrt{2} - 2/3 \end{bmatrix}$$

And get the covariance Matrix:

$$\begin{aligned} C &= 1/3 \cdot X_C^T X_C \\ &= 1/3 \cdot \begin{bmatrix} 44/3 & 40/3 \\ 40/3 & 56/3 \end{bmatrix} \end{aligned}$$

The actual expanded form of this with appropriate steps in between would be way too long to fit on one page.
We actually don't care for the scalar 1/9, since the eigenvalues are independent of it since the characteristic polynomial would just be scaled by a scalar, so the zeroes of it are the same.
With this, to get the eigenpairs of the covariance matrix C, we can just get the eigenpairs of $9 \cdot C$, since it does not matter, since we only need the eigenvectors and they will be normalized later anyways.
We have to solve:

$$\begin{aligned} 0 &= \det(9 \cdot C - \lambda I_{2\times 2}) \\ &= \det\left(\begin{bmatrix} 44 - \lambda & 40 \\ 40 & 56 - \lambda \end{bmatrix}\right) \\ &= (44 - x)(56 - x) - 40 \cdot 40 \\ &= x^2 - 100x + 864 \end{aligned}$$

This polynomial has solutions:

$$\lambda_1 = 50 + 2\sqrt{409} \wedge \lambda_2 = 50 - 2\sqrt{409}$$

With λ_1 being the bigger eigenvalue, we know that the direction of biggest covariance is the eigenvector corresponding to λ_1.
On a side note, the eigenvector corresponding to λ_2 must be orthogonal to the one corresponding to λ_1, since C is symmetric, this also makes more sense regarding what were calculating.
To get λ_1's eigenvector, we have to solve:

$$\begin{aligned}\begin{bmatrix}0\\0\end{bmatrix} &= \begin{bmatrix}44-\lambda_1 & 40\\ 40 & 56-\lambda_1\end{bmatrix} \times \begin{bmatrix}v_x\\v_y\end{bmatrix}\\ &= \begin{bmatrix}-(6+2\sqrt{409})v_x + 40v_y\\ 40v_x + (6-2\sqrt{409})v_y\end{bmatrix}\end{aligned}$$

Which has infinite solutions, one being:

$$v = \begin{bmatrix}1\\ 1/20(\sqrt{409}-3)\end{bmatrix}$$

Which is our eigenvector. And with that being the direction of maximal covariance in X.

On a side note, you see all calculation in this subtask thus far would have been trivial, if X would have been centered already to begin with, since then we could just have took the first eigenvector in V from 1.1,this whole ordeal makes the hint „you may use your results from one subtask for the other" extremely misleading.

Now we have[2] to normalize v. The length of v is $s := \sqrt{1 + 1/400 \cdot (\sqrt{409}-3)^2} \approx 1.31971352$, which has no further simplification.
Finally we get:

$$v_n = \frac{1}{s} \cdot \begin{bmatrix}1\\ 1/20(\sqrt{409}-3)\end{bmatrix}$$

As the normalized version.
According to the lecture, the transformation matrix that projects any point onto the corresponding line is $v_n \times v_n^T = s^{-2} \cdot v \times v^T$ which is:

$$\begin{bmatrix}\frac{1}{2} + \frac{3}{2\sqrt{409}} & \frac{10}{\sqrt{409}}\\ \frac{10}{\sqrt{409}} & \frac{1}{2} - \frac{3}{2\sqrt{409}}\end{bmatrix}$$

Which is approximately:

$$\begin{bmatrix}0.57417 & 0.49447\\ 0.49447 & 0.42583\end{bmatrix}$$

□

[2] I think we don't have to, since for $s \in \mathbb{R}$ we have $(s \cdot u) \times (s \cdot u^T) = s^2 \cdot u \times u^T$, which means that the projected point would be still on the line, but just with different distance to (0,0), but the lecture suggests that we should.

2 TF-IDF

1. Vocabulary: [”fast”, ”car”, ”highway”, ”road”, ”bike”, ”wheel”]

2. term frequencies:

t	$d = 1$	$d = 2$	$d = 3$	$d = 4$
fast	0.2	0.4	0.2	0
car	0.4	0.4	0	0.25
highway	0.2	0	0.2	0
road	0.2	0	0.4	0
bike	0	0.2	0	0.25
wheel	0	0	0.2	0.5

3. document frequency:

t	$\#d$ containing t	IDF
fast	3	$\log \frac{4}{3}$
car	3	$\log \frac{4}{3}$
highway	2	$\log \frac{4}{2} = \log 2$
road	2	$\log(2)$
bike	2	$\log(2)$
wheel	2	$\log(2)$

4. calculating TF-IDF vectors:

t	$d = 1$	$d = 2$	$d = 3$	$d = 4$
fast	0.025	0.05	0.025	0
car	0.05	0.05	0	0.031
highway	0.06	0	0.06	0
road	0.06	0	0.120	0
bike	0	0.06	0	0.075
wheel	0	0	0.06	0.151

Which gives us the following vectors for each d:

$$
\begin{aligned}
r(d = 1) &= (0.025, 0.05, 0.06, 0.06, 0, 0) \\
r(d = 2) &= (0.05, 0.05, 0, 0, 0.06, 0) \\
r(d = 3) &= (0.025, 0, 0.06, 0.12, 0, 0.06) \\
r(d = 4) &= (0, 0.031, 00, 0.075, 0.151)
\end{aligned}
$$

Network_Embedding

December 19, 2024

[294]:
```
import numpy as np
import matplotlib.pyplot as plt
from IPython import display
```

Christian-Albrechts-Universität zu Kiel – Institut für Informatik – Marine Data Science

WiSe 2024/25 Lecture: Data Science

Prof. Dr. Kevin Köser	kk@informatik.uni-kiel.de

Network Embedding

0.1 Global Path Planning

Graph-search algorithms like Dijkstra or A∗ can be used to plan paths in graphs from a start to a goal. If the cells of a grid map are represented as vertices of a graph with edges between the neighboring cells, graph-search algorithms can be used for robot path planning. For this exercise sheet we consider the 8-neighborhood of a cell $< x, y >$, which is defined as the set of cells that are adjacent to $< x, y >$ either horizontally, vertically or diagonally.

You can find an implementation of graph-based 2D path planning-framework at the end this notebook. Complete the missing pieces following the instructions below.

0.2 Dijkstra Algorithm

The Dijkstra algorithm can be used to calculate minimum cost paths in a graph. During search, it always chooses the vertex from the graph with the lowest cost from the start and adds its neighboring vertices to the search graph.

- Let M (x, y) denote an occupancy grid map. During search, the grid cells are connected to their neighboring cells to construct the search graph. The function get_neighborhood takes the coordinates of a cell and returns a n × 2 vector with the cell coordinates of its neighbors, considering the boundaries of the map.

[295]:
```
def get_neighborhood(cell, occ_map_shape):
    '''
    Arguments:
```

```
    cell -- cell coordinates as [x, y]
    occ_map_shape -- shape of the occupancy map (nx, ny)

    Output:
    neighbors -- list of up to eight neighbor coordinate tuples [(x1, y1), (x2,␣
↪y2), ...]
    '''

    neighbors = []

    for i in [-1, 0, 1]:
      for j in [-1, 0, 1]:
        x = cell[0] + i
        y = cell[1] + j
        if x < 0 or x >= occ_map_shape[0]: continue
        if y < 0 or y >= occ_map_shape[1]: continue
        if i == 0 and j == 0: continue
        neighbors.append((x, y))


    return neighbors
```

- get_edge_cost formulates a function for the edge costs between two cells that allows for planning of the shortest collision free path on the grid. It include occupancy information to prefer cells with low occupancy probability over cells with higher probability. We regard a cell as an obstacle if its occupancy probability exceeds a threshold of 0.5.

[296]:
```
def get_edge_cost(parent, child, occ_map):
    '''
    Calculate cost for moving from parent to child.

    Arguments:
    parent, child -- cell coordinates as [x, y]
    occ_map -- occupancy probability map

    Output:
    edge_cost -- calculated cost
    '''

    edge_cost = 0

    occ = occ_map[child[0], child[1]]
    if occ >= .5:
      return np.inf
    edge_cost = np.linalg.norm(parent - child)

    return edge_cost + 10 * occ
```

- The update step of the Dijkstra algorithm in run_path_planning. For the current parent node, we consider all of its neighbors and calculate their tentative distances from the start location (cost) and their predecessor in the grid. You are now ready to run the Dijkstra algorithm within the planning_framework.*

[297]:
```
def update_step(x, y, costs, predecessors, parent, occ_map):
    '''
    In-place update of costs of predecessors for neighbours

    Arguments:
    x,y -- cell coordinates of child node
    costs -- costs[x,y] is the minimum currently know distance to cell at (x,y)
    parent -- cell coordinates as [x, y]
    predecessors -- predecessors[x,y] is the predecessors to cell at (x,y) on␣
 ↪the best known path
    occ_map -- occupancy probability map

    Output:


    '''
    neighbors = get_neighborhood(parent, occ_map.shape)
    for child in neighbors:
        child_cost = costs[x, y] + get_edge_cost(parent, child, occ_map)
        if child_cost < costs[child]:
            costs[child] = child_cost
            predecessors[child] = parent
```

0.3 Exercise: Reference Point Embedding

The A∗ algorithm employs a heuristic to perform an informed search with higher efficiency than the Dijkstra algorithm.

- **(a)** *Define a heuristic for optimal 2D mobile robot path planning. Complete the function get_heuristic in the planning framework. The function takes the coordinates of a cell and the goal and returns the estimated costs to the goal. You are now ready to run the A∗ algorithm with the planning_framework.*

The Euclidean distance gives a lower bound for the cost and therefore is a good heuristic. It is equivalent to the actual cost in case of a straight path along cells with zero cost.

- **(b)** *What happens if you inflate your heuristic by using h2 , which is a multiple of your defined heuristic h? Try different multiples:* $h2 = 1, 2, 5, 10 \cdot h$

0.3.1 Answer:

- **(c)** Implement the reference-based graph embedding by filling in the stub of the function `calculate_embedding`.

- This function should compute embeddings for the graph nodes using the reference-node based approach.

- **(d)** Fill in the stub function `calculate_network_distance_estimations`.
  - This function should compute a lower bound of the network distance between nodes, utilizing the embedding generated in **Task (c)**.

[298]:
```python
def get_heuristic(cell, goal, embedding = None):
    '''
    Estimate cost for moving from cell to goal based on heuristic.
    Arguments:
    cell, goal -- cell coordinates as [x, y]
    embedding -- optional. Pass the reference node embedding if network␣
↪distance based estimation is to be applied
    Output:
    cost -- estimated cost
    '''

    def calculate_network_distance_estimations(embedding, cell, goal):
        ##########################################
        # task d
        ''' to be quite honest, I have little to no idea what's the goal here
        the network based algorithm now produces a valid path - it may not be
        the shortest and certainly isn't the most cost efficient, but it's a␣
↪path.'''
        cell = tuple(cell)
        goal_tuple = tuple(goal)

        cell_embedding = embedding[cell]
        goal_embedding = embedding[goal_tuple]
        return sum(abs(c-g) for c,g in zip(cell_embedding, goal_embedding))
        ##########################################
        return 0

    heuristic = 0
    h2 = 1

    # network distance based
    if embedding is not None:
        heuristic = h2 * calculate_network_distance_estimations(embedding,␣
↪cell, goal)

    # linear distance based
    else:
        ##########################################
        x1, y1 = cell
        x2, y2 = goal
        heuristic = ((x1-x2)**2 + (y1-y2)**2)**0.5
```

```
        '''
        With larger h2 multipliers the algorithm gets significantly faster at a␣
↪cost
        of accuracy - the computed path might not be the most efficient.
        '''
        ##########################################
        pass

    return heuristic
```

[ ]:
```
def calculate_embedding(no_of_reference_nodes, occ_map):

  def shortest_path(ref_node, occ_map):
    # cost values for each cell, filled incrementally.
    # Initialize with infinity
    costs = np.ones(occ_map.shape) * np.inf

    # cells that have already been visited
    closed_flags = np.zeros(occ_map.shape)

    # store predecessors for each visited cell
    predecessors = -np.ones(occ_map.shape + (2,), dtype=int)

    # start search
    parent = ref_node
    costs[ref_node[0], ref_node[1]] = 0

    # loop until goal is found
    while True:

      # costs of candidate cells for expansion (i.e. not in the closed list)
      open_costs = np.where(closed_flags==1, np.inf, costs)

      # find cell with minimum cost in the open list
      x, y = np.unravel_index(open_costs.argmin(), open_costs.shape)

      # break loop if minimal costs are infinite (no open cells anymore)
      if open_costs[x, y] == np.inf:
        break

      # set as parent and put it in closed list
      parent = np.array([x, y])
      closed_flags[x, y] = 1

      # update costs and predecessor for neighbors
      update_step(x, y, costs, predecessors, parent, occ_map)
```

```
    return costs

    #choose n random reference points


  ########################################

  # task c

  accessible_cells = np.argwhere(occ_map == 0)
  ref_nodes = accessible_cells[np.random.choice(len(accessible_cells),␣
↪no_of_reference_nodes, replace = False)]

  embedding = {}   # store embeddings in dict
  for ref_node in ref_nodes:
    cost = shortest_path(ref_node, occ_map)
    for x in range(occ_map.shape[0]):
      for y in range(occ_map.shape[1]):
        #if occ_map[x,y] == 0:       # only accessible cells
          if (x,y) not in embedding:
            embedding[(x,y)] = []    # embedding for each node stored in list
          embedding[(x,y)].append(cost[x,y])
  return embedding
  ########################################

  #inaccessible cells should not be reference points


  return 0
```

1 Path Planning Framework

[300]:
```
def plot_map(occ_map, start, goal):

  plt.imshow(occ_map.T, cmap=plt.cm.gray, interpolation='none', origin='upper')
  plt.plot([start[0]], [start[1]], 'ro')
  plt.plot([goal[0]], [goal[1]], 'go')
  plt.axis([0, occ_map.shape[0]-1, 0, occ_map.shape[1]-1])
  plt.xlabel('x')
  plt.ylabel('y')

def plot_expanded(expanded, occ_map, start, goal, clear=True):

  if np.array_equal(expanded, start) or np.array_equal(expanded, goal):
    return
```

```
  plot_map(occ_map, start, goal)
  idx = expanded.nonzero()

  plt.plot(idx[0], idx[1], 'yo')
  #plt.show()
  plt.plot([start[0]], [start[1]], 'ro')
  plt.plot([goal[0]], [goal[1]], 'go')
  if clear:
    display.clear_output(wait=True)
    plt.pause(1e-8)

def plot_path(expanded, occ_map, path, start, goal):

  if np.array_equal(path, goal):
    return

  plot_map(occ_map, start, goal)
  plot_expanded(expanded, occ_map, start, goal, False)
  x = [xy[0] for xy in path]
  y = [xy[1] for xy in path]
  plt.plot(x, y, 'bo')
  plt.plot([goal[0]], [goal[1]], 'go')

  display.clear_output(wait=True)
  plt.pause(.01)

def plot_costs(cost):
  plt.figure()
  plt.title("Cost")
  plt.imshow(cost.T, cmap=plt.cm.gray, interpolation='none', origin='upper')
  plt.axis([0, cost.shape[0]-1, 0, cost.shape[1]-1])
  plt.xlabel('x')
  plt.ylabel('y')


def run_path_planning(occ_map, start, goal, h2_multiplier = 1,␣
 ↪use_heuristic=None):
  '''
  This implements the
  - A* algorithm using linear distance (in case heuristic is "linear_distance")
  - A* algorithm using reference node based network distance (in case heuristic␣
 ↪is "network_based")
  - Dikstra algorithm (in case heuristic is none of the above)
  '''

  plot_map(occ_map, start, goal)
```

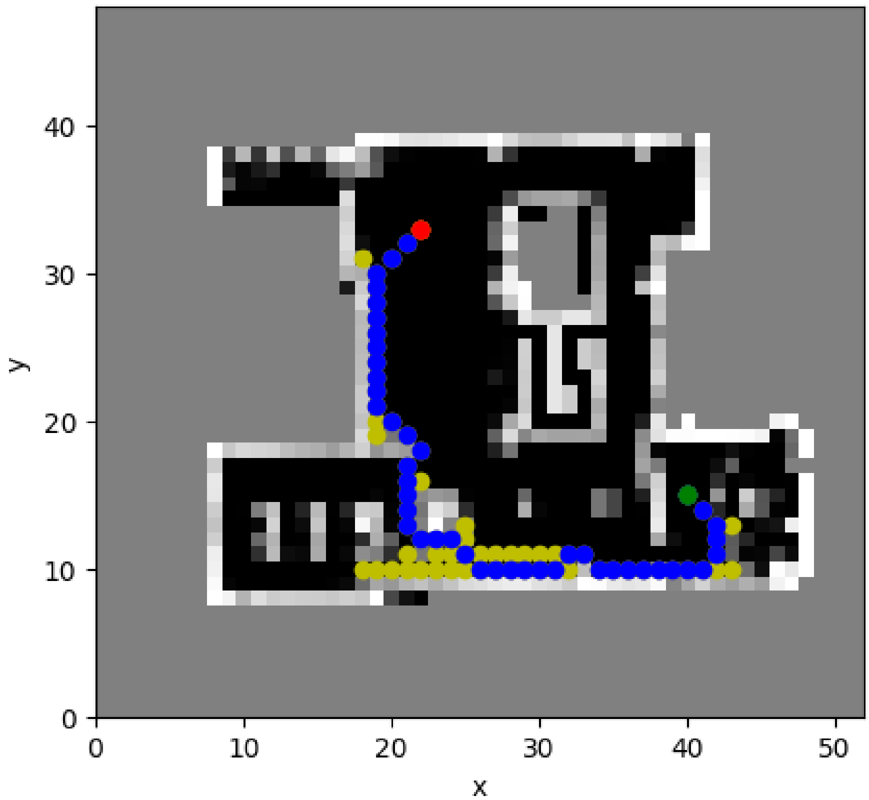
```
# cost values for each cell, filled incrementally.
# Initialize with infinity
costs = np.ones(occ_map.shape) * np.inf

# cells that have already been visited
closed_flags = np.zeros(occ_map.shape)

# store predecessors for each visited cell
predecessors = -np.ones(occ_map.shape + (2,), dtype=int)


# heuristic for A*
heuristic = np.zeros(occ_map.shape)

# linear distance heuristic
if use_heuristic == "linear_distance":
  for x in range(occ_map.shape[0]):
    for y in range(occ_map.shape[1]):
      heuristic[x, y] = get_heuristic([x, y], goal)

# network distance heuristic usding reference nodes
elif use_heuristic == "network_based":
  no_of_reference_nodes = 10
  embedding = calculate_embedding(no_of_reference_nodes, occ_map)
  for x in range(occ_map.shape[0]):
    for y in range(occ_map.shape[1]):
      heuristic[x, y] = get_heuristic([x, y], goal, embedding)

#Dijkstra - no heuristic
else:
  pass

# start search
parent = start
costs[start[0], start[1]] = 0

# loop until goal is found
while not np.array_equal(parent, goal):

  # costs of candidate cells for expansion (i.e. not in the closed list)
  open_costs = np.where(closed_flags==1, np.inf, costs) + heuristic

  # find cell with minimum cost in the open list
  x, y = np.unravel_index(open_costs.argmin(), open_costs.shape)

  # break loop if minimal costs are infinite (no open cells anymore)
```

```python
    if open_costs[x, y] == np.inf:
      break

    # set as parent and put it in closed list
    parent = np.array([x, y])
    closed_flags[x, y] = 1

    # update costs and predecessor for neighbors
    update_step(x, y, costs, predecessors, parent, occ_map)

    #visualize grid cells that have been expanded
    plot_expanded(closed_flags, occ_map, start, goal)

  # rewind the path from goal to start (at start predecessor is [-1,-1])
  if np.array_equal(parent, goal):
    path_length = 0
    path = []
    while predecessors[parent[0], parent[1]][0] >= 0:

      predecessor = predecessors[parent[0], parent[1]]
      path_length += np.linalg.norm(parent - predecessor)
      path.append(parent)
      parent = predecessor

      plot_path(closed_flags, occ_map, path, start, goal)

    print ("found goal     : " + str(parent) )
    print ("cells expanded : " + str(np.count_nonzero(closed_flags)) )
    print ("path cost      : " + str(costs[goal[0], goal[1]]) )
    print ("path length    : " + str(path_length) )
  else:
    print ("no valid path found")

  #plot the costs
  plot_costs(costs)
```

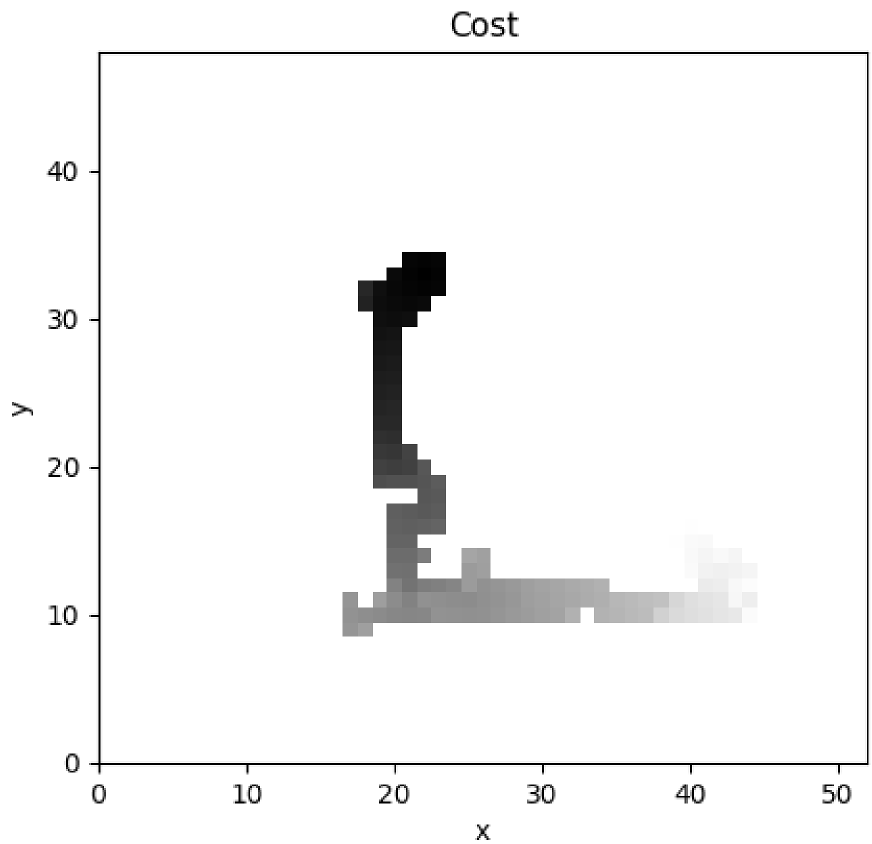
[ ]:
```python
#run Dijkstra

# load the occupancy map
occ_map = np.loadtxt('map.txt')

# start and goal position [x, y]
start = np.array([22, 33])
goal = np.array([40, 15])

run_path_planning(occ_map, start, goal)#, use_heuristic="linear_distance")
```

[ ]:
```
#run A* with linear distance heuristic
run_path_planning(occ_map, start, goal, use_heuristic="linear_distance")
```

[303]:
```
#run A* with network distance based heuristic
run_path_planning(occ_map, start, goal, use_heuristic="network_based")
```




```
found goal     : [22 33]
cells expanded : 73
path cost      : 70.32856299236305
path length    : 51.21320343559641
```


Cost
y
40
30
20
10
0
0
10
20
30
40
50
x