Simulation Step 4 Finding the Shortest Delivery Route

March 7, 2024

Table of Contents

[1]:
```
import matplotlib.pyplot as plt
import pulp
import math
import random
import pandas as pd
import numpy as np
```

1 Utilities (Copied from Starter File)

1.1 Points and Distances

[2]:
```
def dist(p1, p2):
    (x1, y1) = p1
    (x2, y2) = p2
    return int(math.sqrt((x1-x2)**2+(y1-y2)**2))
```

1.2 PlotMap

[3]:
```
def plotMap(G, T=[], P=[], W=None,
            style='r-o', lw=1, ms=3,
            styleT='go', msT=5,
            styleP='b-o', lwP=3, msP=1,
            stylePT='go', msPT=7,
            styleW='bo', msW=9,
            text=None, grid=False):
    fig = plt.gcf()
    fig.set_size_inches(6, 6)
    V, E = G

    if not grid:
        plt.axis('off')
    plt.plot( [ p[0] for p in V ], [ p[1] for p in V ], 'ro', lw=lw, ms=ms)
    for (p, q) in E:
        plt.plot( [ p[0], q[0] ], [ p[1], q[1] ], 'r-o', lw=lw, ms=ms)
    for t in T:
        plt.plot( [ t[0] ], [ t[1] ],
                  styleT, ms=msT)
    plt.plot( [ p[0] for p in P ],
              [ p[1] for p in P ],
             styleP, lw=lwP, ms=msP)
    for p in P:
        if p in T:
            plt.plot( [ p[0] ], [ p[1] ],
                      stylePT, ms=msPT)
    if W is not None:
        plt.plot( [ W[0] ], [ W[1] ],
                      styleW, ms=msW)
    if text is not None:
        maxX = max([p[0] for p in V])
        plt.text(0.8*maxX, 0, text)
    if grid:
        plt.grid()
    plt.show()
```

1.3 Add Targets

[4]:
```
def addTargets(M, T):
    V, E = M
    E = E.copy()
    V = V.copy()
    for t in T:
        minD = math.inf
        minE = None
        for e in E:
            P, Q = e
            distT = dist(P, t)+dist(t, Q)-dist(P, Q)
            if distT < minD:
                minD = distT
                minE = e
        P, Q = minE
        E.remove( (P, Q) )
        E.append( (P, t) )
        E.append( (t, Q) )
        V.append(t)
    return V, E
```

1.4 Generate Warehouse Location

This is a blind random generation as it would be needed for a Monte-Carlo Optimisation. You may improve this algorithm to reduce the search space.

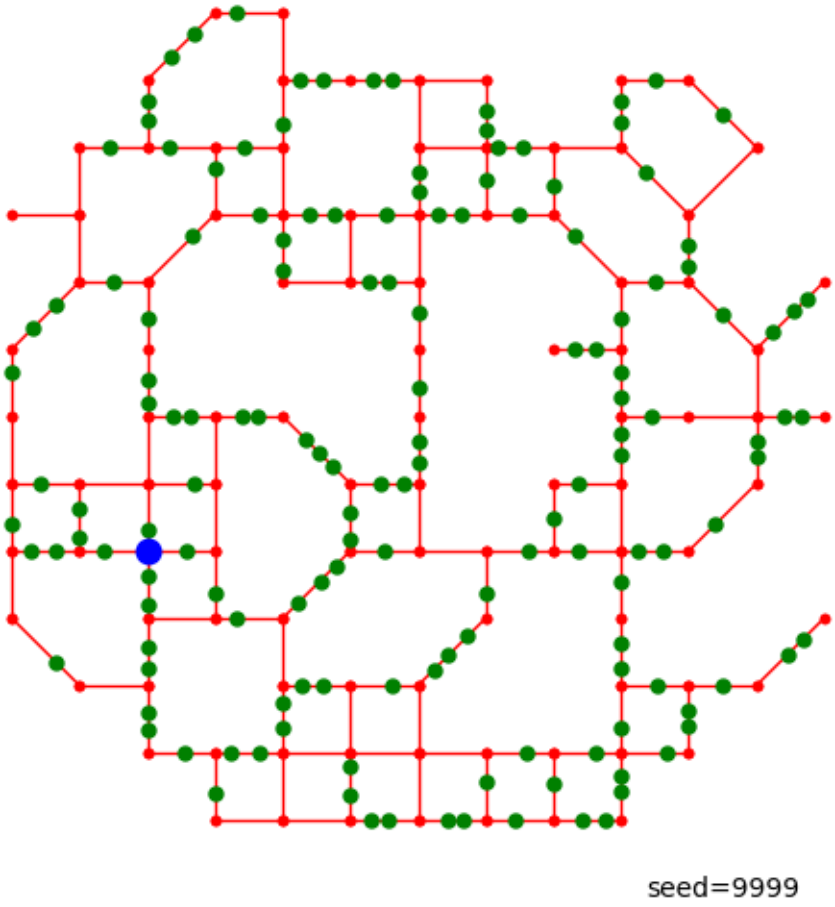
[5]:
```
def generateWarehouseLocation(M):
    V, _ = M
    W = random.sample(V, k=1)[0]
    return W
```

2 Load Pickled Sample Data

[6]:
```
import pickle
with open('data.pickled', 'rb') as f:
    M, C = pickle.load(f)
```

[7]:
```
random.seed(9999)
W = generateWarehouseLocation(M)
```

[8]:
```
plotMap(M, T=C, P=[], W=W, text="seed=9999")
```



3 Finding the Shortest Past

3.1 The Algorithm

This is the A^* algorithm introduced in Week 3.

[9]:
```
def pathLength(P):
    return 0 if len(P)<=1 else \
            dist(P[0], P[1])+pathLength(P[1:])
```

[10]:
```
def shortestPath(M, A, B):

    def h(p):
        return pathLength(p)+dist(p[-1],B)

    # candidates C are pairs of the path so far and
    # the heuristic function of that path,
```

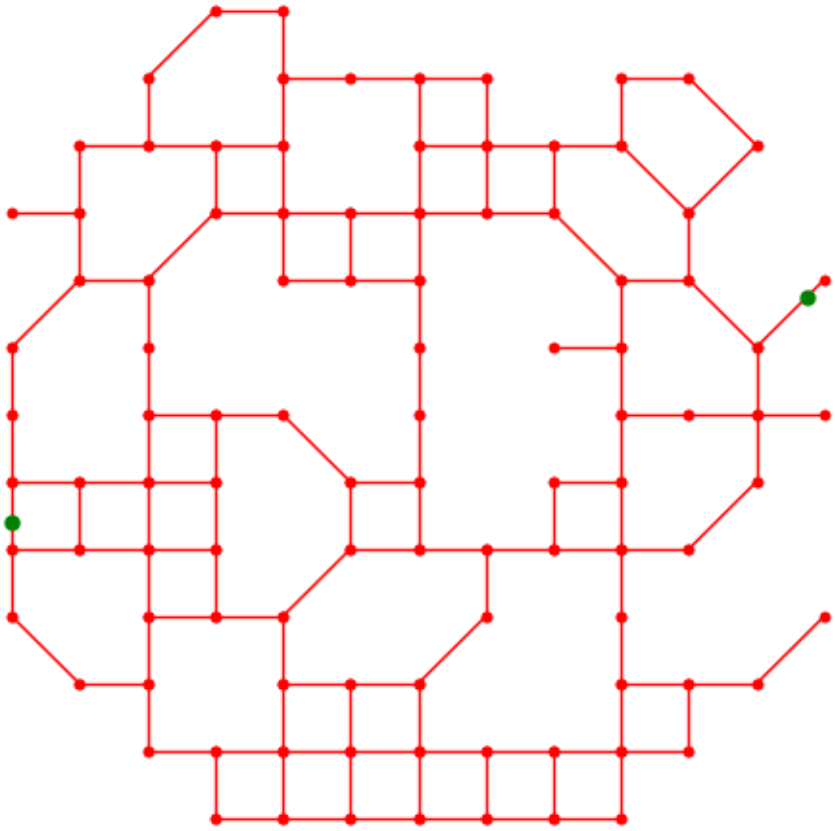
```
    # sorted by the heuristic function, as maintained by
    # insert function
    def insert(C, p):
        hp = h(p)
        c = (p, hp)
        for i in range(len(C)):
            if C[i][1]>hp:
                return C[:i]+[c]+C[i:]
        return C+[c]

    V, E = M
    assert(A in V and B in V)
    C = insert([], [A])

    while len(C)>0:
        # take the first candidate out of the list of candidates
        path, _ = C[0]
        C = C[1:]
        if path[-1]==B:
            return path
        else:
            for (x, y) in E:
                if path[-1]==x and y not in path:
                    C = insert(C, path+[y])
                elif path[-1]==y and x not in path:
                    C = insert(C, path+[x])
    return None
```

3.2 Testing

[11]:
```
A = C[0]
B = C[-1]
```

[12]:
```
MAB = addTargets(M, [A, B])
```

[13]:
```
plotMap(MAB, T=[A, B])
```



[14]:
```
P = shortestPath(MAB, A, B)
```

[15]:
```
P
```

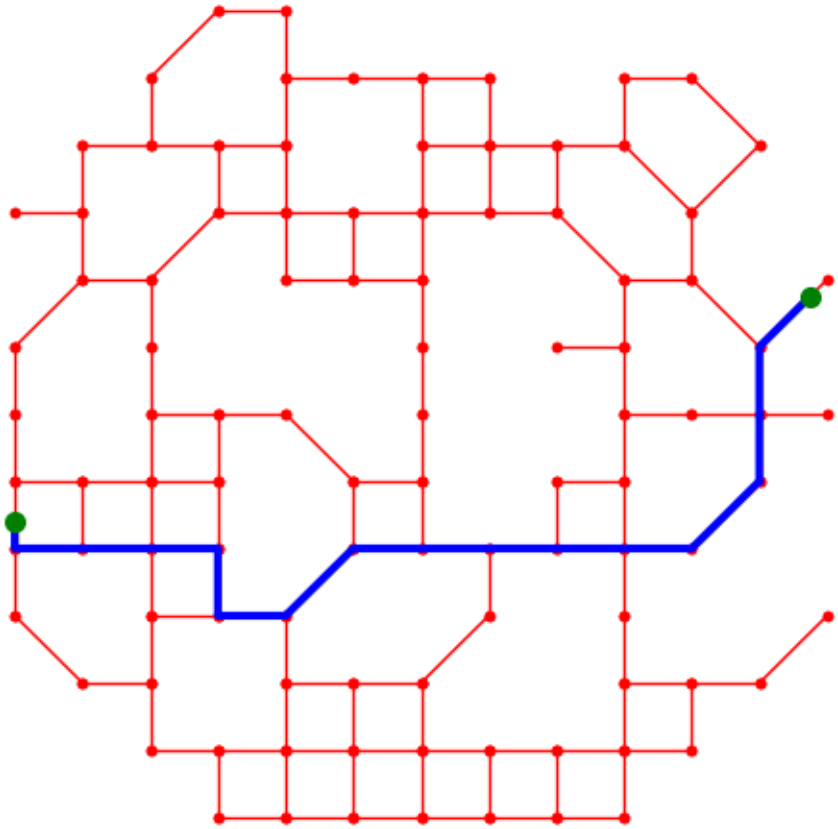
[15]:
```
[(640, 3104),
 (640, 2880),
 (1200, 2880),
 (1760, 2880),
 (2320, 2880),
 (2320, 2320),
 (2880, 2320),
 (3440, 2880),
 (4000, 2880),
 (4560, 2880),
 (5120, 2880),
 (5680, 2880),
 (6240, 2880),
 (6800, 3440),
```

```
 (6800, 4000),
 (6800, 4560),
 (7214, 4974)]
```

[16]:
```
pathLength(P)
```

[16]: 9111

[17]:
```
plotMap(MAB, T=[A, B], P=P)
```



4 Finding Shortest Delivery Route

4.1 Iterative Integer Programming

[18]:
```
def createTables(M, T):

    def reverse(P):
        return [ P[-i] for i in range(1,len(P)+1) ]
```

```
    def index(x, L):
        for i in range(len(L)):
            if x==L[i]:
                return i
        return None

    n = len(T)
    d = [ [ math.inf for t in T ] for t in T ]
    p = [ [ None for t in T ] for t in T ]
    for i in range(n):
        d[i][i] = 0
        p[i][i] = [ T[i] ]
    for i in range(n):
        for j in range(n):
            if p[i][j] is None:
                s = shortestPath(M, T[i], T[j])
                d[i][j] = d[j][i] = pathLength(s)
                p[i][j] = s
                p[j][i] = reverse(s)
                for m in range(len(s)-1):
                    smi = index(s[m], T)
                    if smi is None:
                        continue
                    for l in range(m+1, len(s)):
                        sli = index(s[l], T)
                        if sli is None:
                            continue
                        sub = s[m:l+1]
                        if p[smi][sli] is None:
                            p[smi][sli] = sub
                            p[sli][smi] = reverse(sub)
                            d[smi][sli] = d[sli][smi] = pathLength(sub)
    return d,p
```

[19]:
```
def roundtrips(x, n):

    def isElem(x, l):
        for i in range(len(l)):
            if l[i]==x:
                return True
        return False

    def startpoint(trips):
        for i in range(n):
            for t in trips:
                if isElem(i, t):
```

```
                    break
            else:
                return i

    def totalLength(trips):
        s=0
        for i in range(0, len(trips)):
            s += len(trips[i])-1
        return s

    trips = []
    while totalLength(trips)<n:
        start = startpoint(trips)
        trip = [ start ]
        i = start
        while len(trip) < n-totalLength(trips):
            for j in range(0, n):
                if pulp.value(x[i][j])==1:
                    trip.append(j)
                    i=j
                    break
            if pulp.value(x[trip[-1]][start])==1:
                trip.append(start)
                break
        trips.append(trip)
    return sorted(trips, key=lambda t: len(t), reverse=True)
```

[20]:
```
import time

def createLoop(M, T, timing=False):

    if timing:
        start_time = time.time()
        last_time = time.time()

    D, P = createTables(M, T)   # These are the distances between customers and␣
 ↪warehouse only

    if timing:
        print(f"createTables:   {time.time()-start_time:6.2f}s")
        last_time = time.time()

    n = len(T)
    # create variables
    x = pulp.LpVariable.dicts("x", ( range(n), range(n) ),
                              lowBound=0, upBound=1, cat=pulp.LpInteger)
    # create problem
```

```
    prob = pulp.LpProblem("Loop",pulp.LpMinimize)
    # add objective function
    prob += pulp.lpSum([ D[i][j]*x[i][j]
                            for i in range(n) for j in range(n) ])
    # add constraints
    constraints=0
    for j in range(n):
        prob += pulp.lpSum([ x[i][j] for i in range(n) if i!=j ]) ==1
    constraints += n
    for i in range(n):
        prob += pulp.lpSum([ x[i][j] for j in range(n) if i!=j ]) ==1
    constraints += n
    for i in range(n):
        for j in range(n):
            if i!=j:
                prob += x[i][j]+x[j][i] <= 1
                constraints += 1
    # initialise solver
    solvers = pulp.listSolvers(onlyAvailable=True)
    solver = pulp.getSolver(solvers[0], msg=0)
    prob.solve(solver)

    if timing:
        print(f"Solver:         {time.time()-last_time:6.2f}s {constraints:6,d}␣
↪Constraints")
        last_time = time.time()

    trips = roundtrips(x, n)
    while len(trips)>1:
        longest = max([ len(t) for t in trips ])
        for t in trips:
            if len(t)<longest:
                prob += pulp.lpSum([ x[t[i]][t[i+1]] + x[t[i+1]][t[i]]
                                        for i in range(0,len(t)-1) ]) <=␣
↪len(t)-2
                constraints += 1
            else:
                longest = math.inf
        prob.solve(solver)

        if timing:
            print(f"Solver:         {time.time()-last_time:6.2f}s {constraints:
↪6,d} Constraints")
            last_time = time.time()

        trips = roundtrips(x, n)
    trip = trips[0]
```

```
    # print(trip)
    loop = []
    for k in range(len(trip)-1):
        sub = P[trip[k]][trip[k+1]]
        loop += sub if len(loop)==0 else sub[1:]

    if timing:
        print(f"createLoop:     {time.time()-start_time:6.2f}s")

    return loop
```

4.2 Heuristic Solution

[21]:
```
def FW(M):

    V, E = M

    n = len(V)
    d = [ [ math.inf for j in range(n) ] for i in range(n) ]
    p = [ [ None for j in range(n) ] for i in range(n) ]

    for (A, B) in E:
        a = V.index(A)
        b = V.index(B)
        d[a][b] = d[b][a] = dist(A, B)
        p[a][b] = [A, B]
        p[b][a] = [B, A]

    for i in range(n):
        d[i][i] = 0
        p[i][i] = [V[i]]

    for k in range(n):
        for i in range(n):
            for j in range(n):
                dk = d[i][k] + d[k][j]
                if d[i][j] > dk:
                    d[i][j] = dk
                    p[i][j] = p[i][k][:-1] + p[k][j]

    return d, p
```

4.2.1 Greedy Algorithm

[22]:
```
def createLoopG(M, T, plot=False, timing=False):

    def makeLoop(L):
        loop = []
        for i in range(len(L)-1):
            A = L[i]
            B = L[i+1]
            a = V.index(A)
            b = V.index(B)
            sub = P[a][b]
            loop += sub if len(loop)==0 else sub[1:]
        return loop

    if timing:
        start_time = time.time()

    V, E = M
    D, P = FW(M)    # note these are the distances between all vertices in M␣
↪(and T)

    if timing:
        print(f"Floyd-Warshall: {time.time()-start_time:6.2f}s")

    W = T[0]
    customers = T[1:]
    if len(T)==1:
        L = T
    elif len(T)<=3:
        L = T + [T[0]]
    else:
        L = T[:3]+[T[0]]
        T = T[3:]
        while len(T)>0:
            if plot:
                loop = makeLoop(L)
                plotMap(M, T=L, P=loop, w=W,
                        grid=True, text=f"{pathLength(loop):,d}m")
            minExt = math.inf
            minInd = None
            selInd = None
            for k in range(len(T)):
                C = T[k]
                c = V.index(C)
                for i in range(0, len(L)-1):
                    A = L[i]
```

```
                    B = L[i+1]
                    a = V.index(A)
                    b = V.index(B)
                    ext = D[a][c] + D[c][b] - D[a][b]
                    if ext<minExt:
                        minExt, minInd, selInd = ext, i+1, k
            L = L[:minInd]+[T[selInd]]+L[minInd:]
            T = T[:selInd]+T[selInd+1:]

    if timing:
        print(f"createLoopG:    {time.time()-start_time:6.2f}s")

    return makeLoop(L)
```

4.2.2 Heuristic Algorithm

This is only the skeleton of the code, i.e. identical to the greedy algorithm given above. Your task is now to look at the application of the heuristic rules 2 and 3 from week O3, and to transsfer the code to improve the quality of the solution.

Rule 2:

Rule 3:

[23]:
```
def createLoopH(M, T, plot=False, timing=False):

    def makeLoop(L):
        loop = []
        for i in range(len(L)-1):
            A = L[i]
            B = L[i+1]
            a = V.index(A)
            b = V.index(B)
            sub = P[a][b]
            loop += sub if len(loop)==0 else sub[1:]
        return loop

    if timing:
        start_time = time.time()

    V, E = M
    D, P = FW(M)    # note these are the distances between all vertices in M␣
↪(and T)

    if timing:
        print(f"Floyd-Warshall: {time.time()-start_time:6.2f}s")

    W = T[0]
```

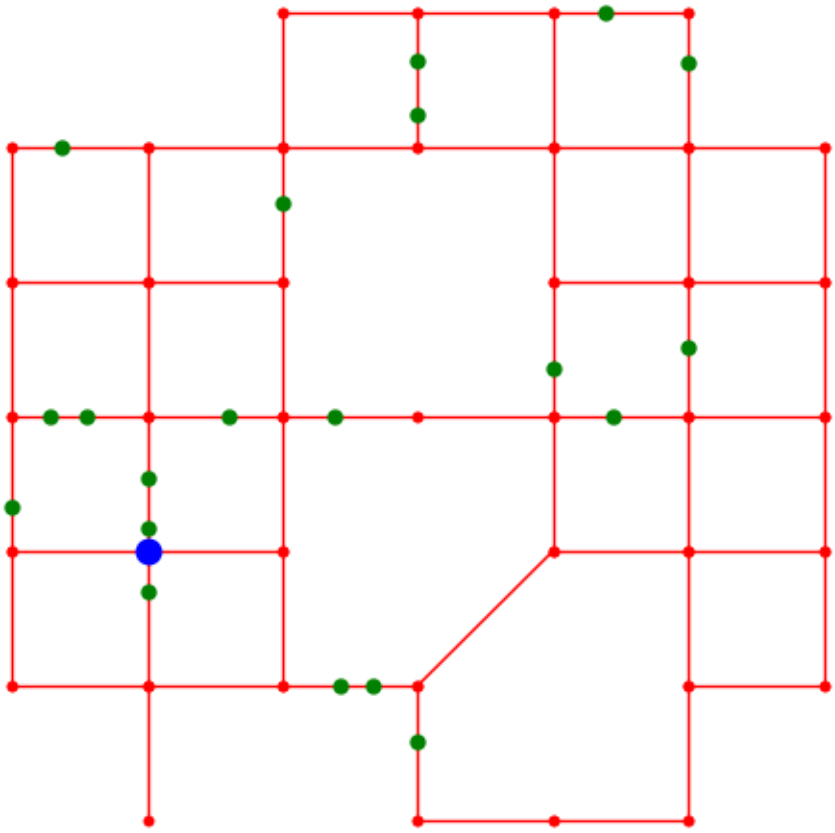
```
    customers = T[1:]
    if len(T)==1:
        L = T
    elif len(T)<=3:
        L = T + [T[0]]
    else:
        L = T[:3]+[T[0]]
        T = T[3:]
        while len(T)>0:
            if plot:
                loop = makeLoop(L)
                plotMap(M, T=L, P=loop, w=W,
                        grid=True, text=f"{pathLength(loop):,d}m")
            minExt = math.inf
            minInd = None
            selInd = None
            for k in range(len(T)):
                C = T[k]
                c = V.index(C)
                for i in range(0, len(L)-1):
                    A = L[i]
                    B = L[i+1]
                    a = V.index(A)
                    b = V.index(B)
                    ext = D[a][c] + D[c][b] - D[a][b]
                    if ext<minExt:
                        minExt, minInd, selInd = ext, i+1, k
            L = L[:minInd]+[T[selInd]]+L[minInd:]
            T = T[:selInd]+T[selInd+1:]

    if timing:
        print(f"createLoopH:    {time.time()-start_time:6.2f}s")

    return makeLoop(L)
```

4.3 Testing

[24]:
```
import pickle
with open('myData.pickled', 'rb') as f:
    M, C = pickle.load(f)
```

[25]:
```
random.seed(42)
W = generateWarehouseLocation(M)
```

[26]:
```
plotMap(M, T=C, P=[], W=W, text="myData")
```



4.3.1 Delivery to 10 Customers

[27]:
```
random.seed(0)
T = random.sample(C, k=len(C)//2)
```

[28]:
```
MC = addTargets(M, T)
```

[29]:
```
P = createLoop(MC, [W]+T)
```

[30]:
```
PG = createLoopG(MC, [W]+T)
```

[31]:
```
PH = createLoopH(MC, [W]+T)
```

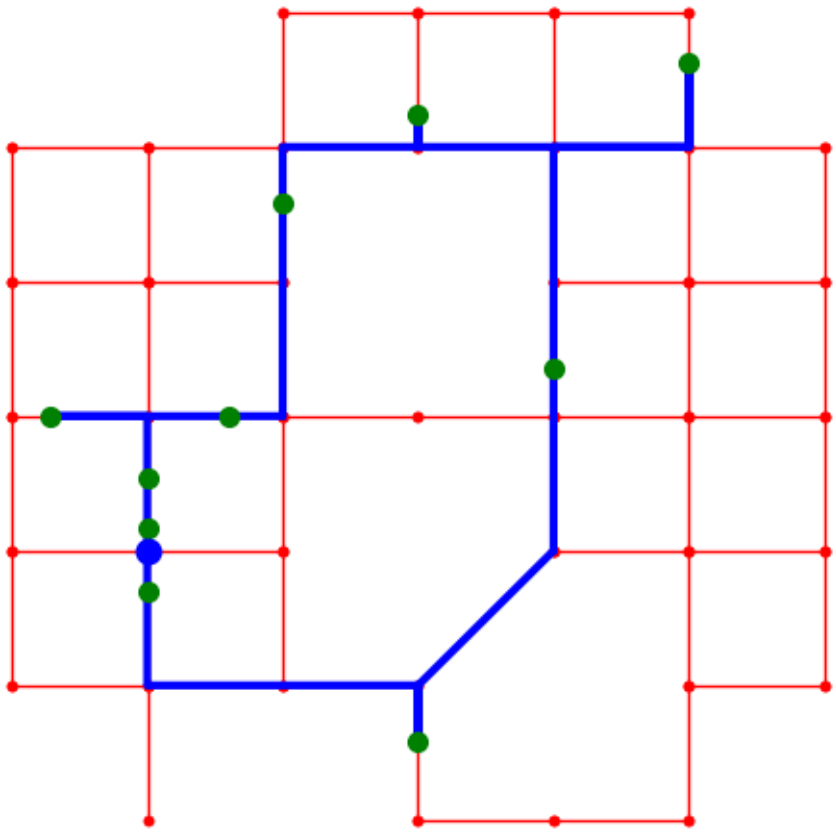
[32]:
```
W
```

[32]:
```
(2240, 3120)
```

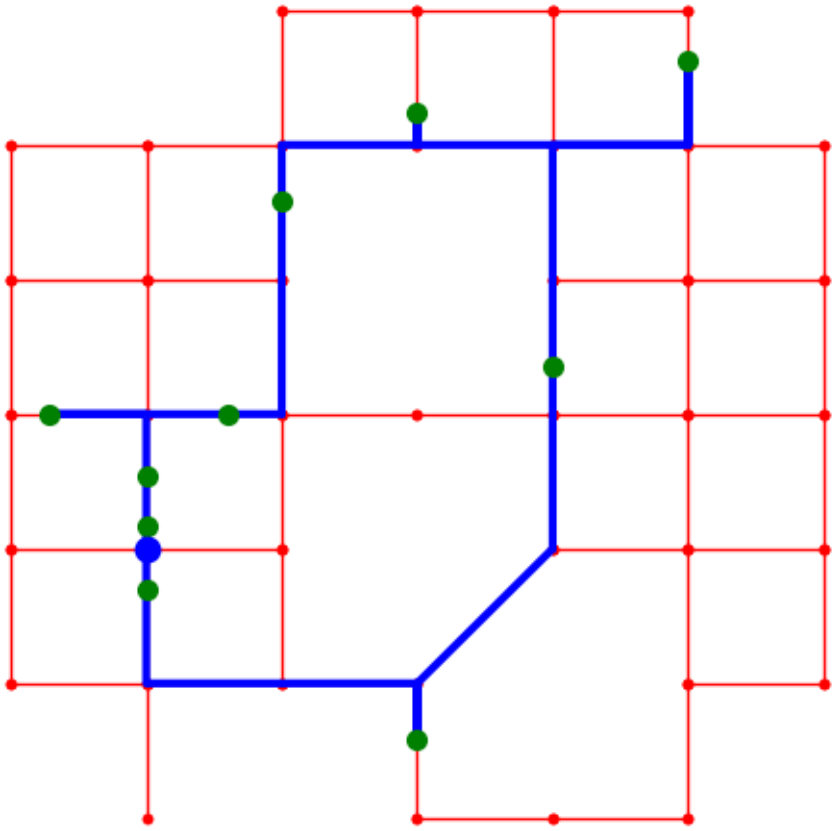
[33]:
```
P
```

[33]:
```
[(2240, 3120),
 (2240, 3268),
 (2240, 3590),
 (2240, 4000),
 (1604, 4000),
 (2240, 4000),
 (2768, 4000),
 (3120, 4000),
 (3120, 4880),
 (3120, 5393),
 (3120, 5760),
 (4000, 5760),
 (4000, 5973),
 (4000, 5760),
 (4880, 5760),
 (5760, 5760),
 (5760, 6317),
 (5760, 5760),
 (4880, 5760),
 (4880, 4880),
 (4880, 4314),
 (4880, 4000),
 (4880, 3120),
 (4000, 2240),
 (4000, 1870),
 (4000, 2240),
 (3120, 2240),
 (2240, 2240),
 (2240, 2853),
 (2240, 3120)]
```

[34]:
```
plotMap(MC, T=T, W=W, P=P, text=f"Path Length={pathLength(P):3,d}m")
```

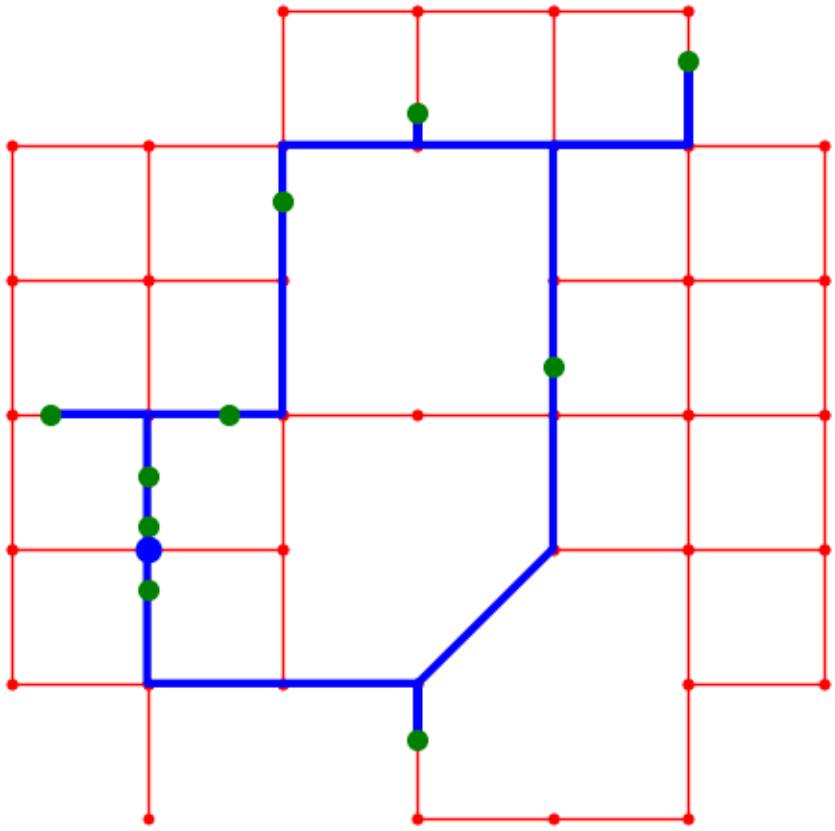


[35]:
```
plotMap(MC, T=T, W=W, P=PG, text=f"Greedy Path Length={pathLength(PG):3,d}m")
```



Greedy Path Length=17,116m

[36]:
```
plotMap(MC, T=T, W=W, P=PH, text=f"Heuristic Path Length={pathLength(PH):3,d}m")
```



Heuristic Path Length=17,116m

[37]:
```
PH.reverse()
```

[38]:
```
P == PH
```

[38]: True

4.3.2 Delivering to all Customers

[39]:
```
T = C
```

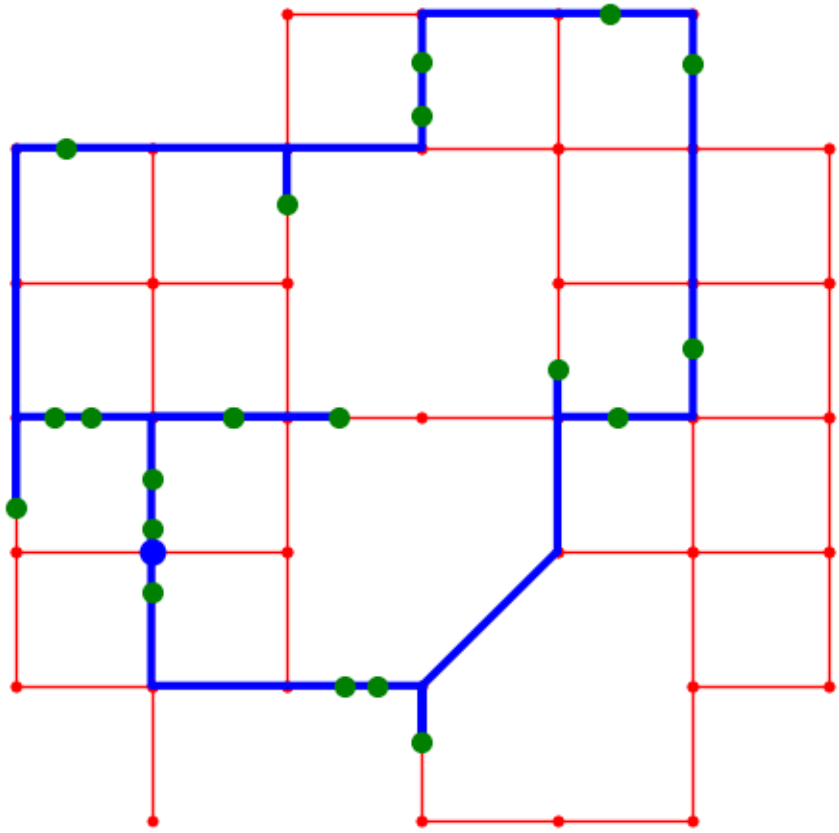
[40]:
```
MC = addTargets(M, T)
```

[41]:
```
PC = createLoop(MC, [W] + T)
```

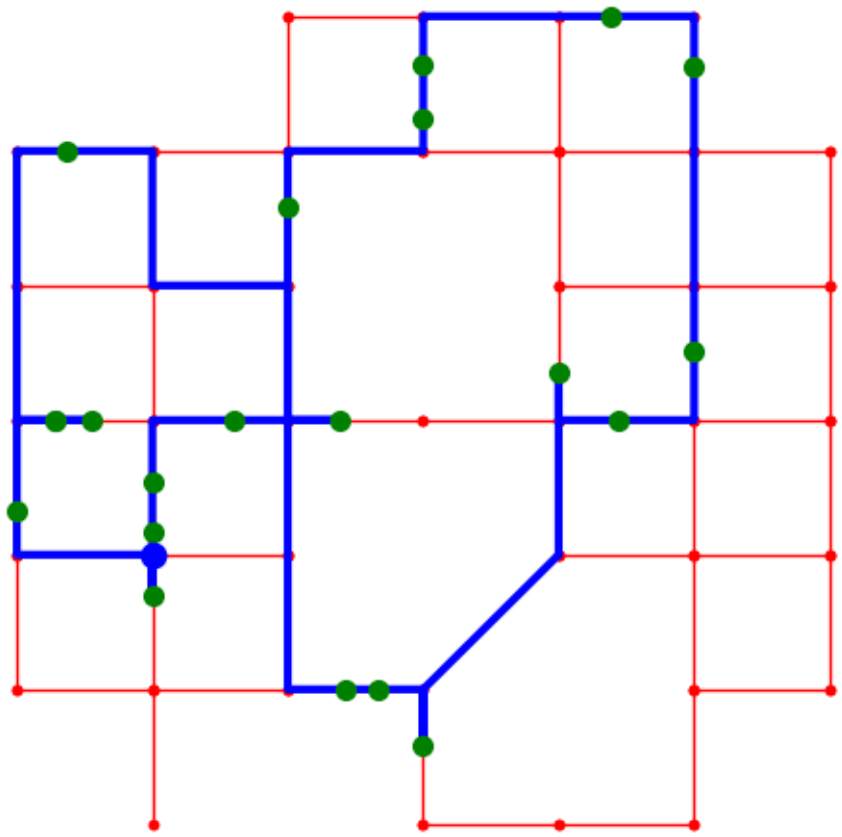
[42]:
```
PCG = createLoopG(MC, [W]+T)
```

[43]:
```
PCH = createLoopH(MC, [W]+T)
```

[44]:
```
plotMap(MC, T=T, W=W, P=PC, text=f"Path Length={pathLength(PC):3,d}m")
```

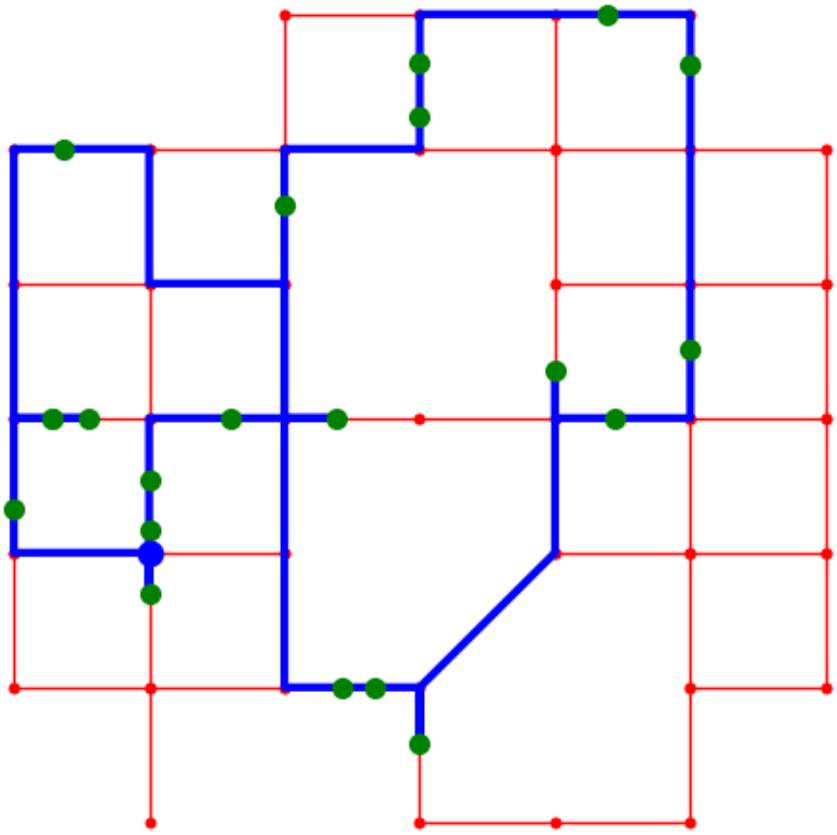


[45]:
```
plotMap(MC, T=T, W=W, P=PCG, text=f"Greedy Path Length={pathLength(PCG):3,d}m")
```



Greedy Path Length=25,928m

[46]:
```
plotMap(MC, T=T, W=W, P=PCH, text=f"Heuristic Path Length={pathLength(PCH):
↪3,d}m")
```



Heuristic Path Length=25,928m

4.4 Running the Algorithm on Real Data

[47]:
```
import pickle
with open('data.pickled', 'rb') as f:
    M, C = pickle.load(f)
```

[48]:
```
random.seed(9999)
W1 = generateWarehouseLocation(M)
```

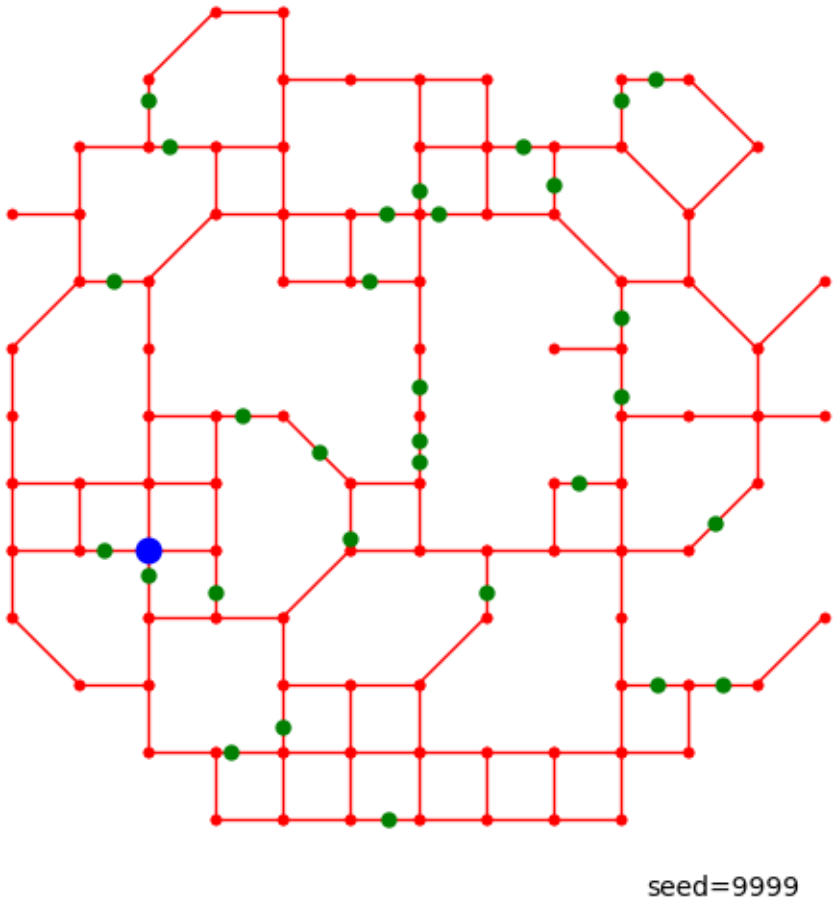
[49]:
```
len(C)
```

[49]: 150

[50]:
```
random.seed(0)
T = random.sample(C, k=len(C)//5)
```

[51]:
```
MT = addTargets(M, T)
```

[52]:
```
plotMap(MT, T=T, W=W1, P=[], text=f"seed=9999")
```



[53]:
```
P1 = createLoop(MT, [W1]+T, timing=True)
```

```
createTables:     0.29s
Solver:           0.20s    992 Constraints
Solver:           0.23s    999 Constraints
Solver:           0.23s  1,004 Constraints
Solver:           0.42s  1,007 Constraints
Solver:           0.80s  1,009 Constraints
createLoop:       2.16s
```

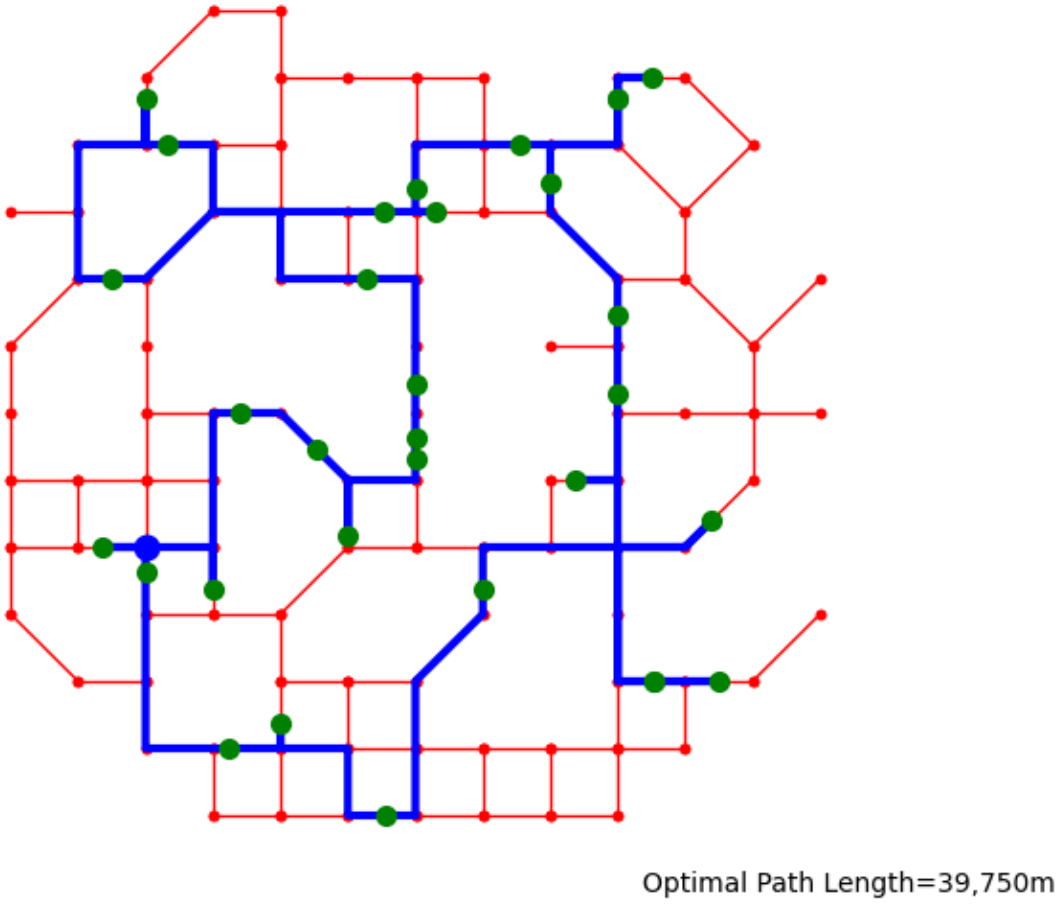
[54]:
```
PG1 = createLoopG(MT, [W1]+T, timing=True)
```

```
Floyd-Warshall:   0.19s
createLoopG:      0.21s
```

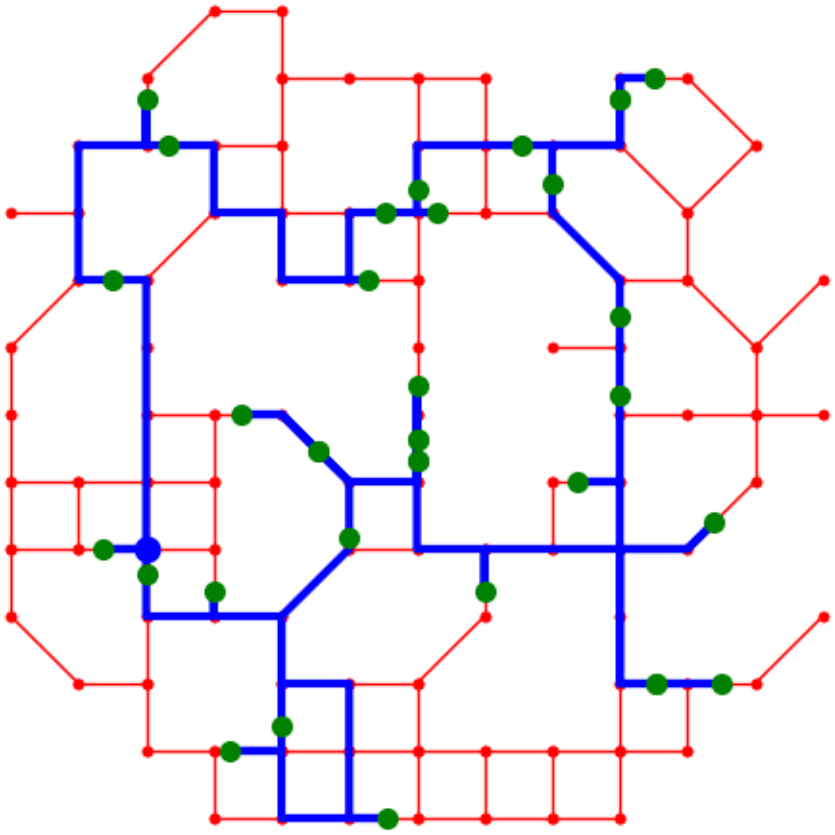
[55]:
```
PH1 = createLoopH(MT, [W1]+T, timing=True)
```

```
Floyd-Warshall:   0.24s
createLoopH:      0.25s
```

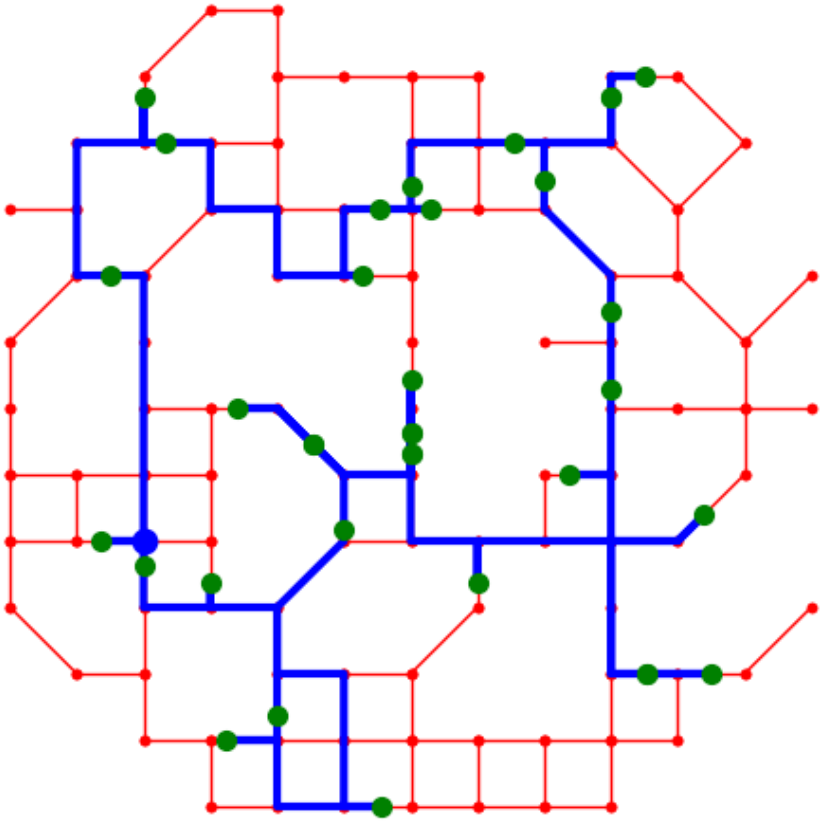
[56]:
```
plotMap(MT, T=T, W=W1, P=P1, text=f"Optimal Path Length={pathLength(P1):3,d}m")
```



[57]:
```
plotMap(MT, T=T, W=W1, P=PG1, text=f"Greedy Path Length={pathLength(PG1):3,d}m")
```

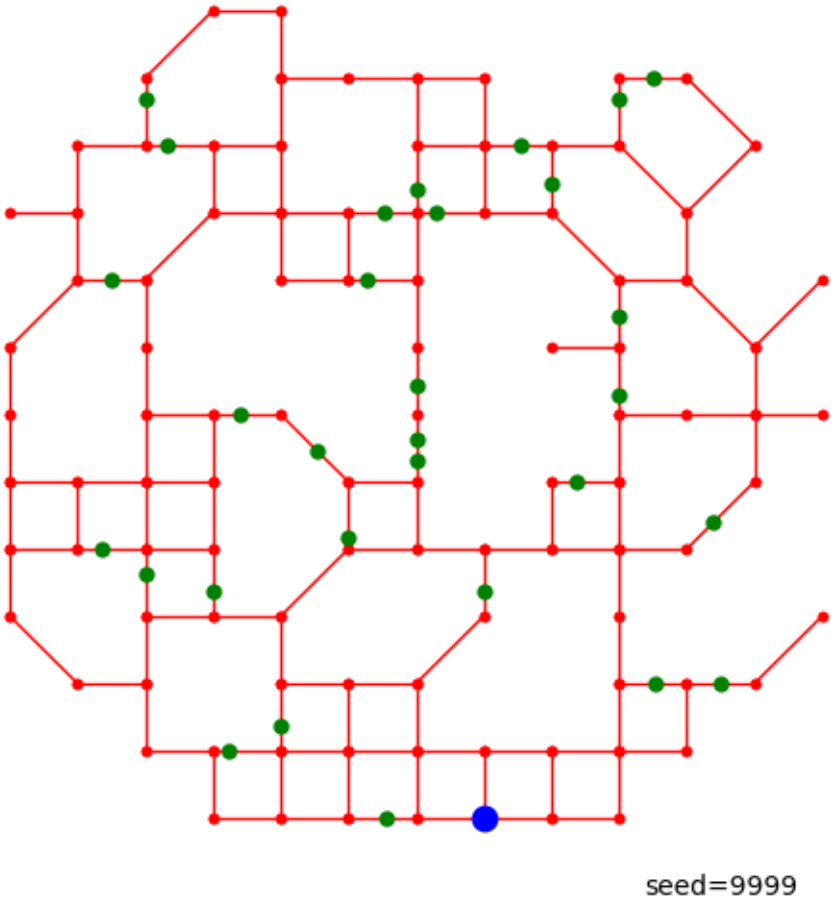


[58]:
```
plotMap(MT, T=T, W=W1, P=PH1, text=f"Heuristic Path Length={pathLength(PH1):
↪3,d}m")
```



[59]:
```
random.seed(12)
W2 = generateWarehouseLocation(M)
```

[60]:
```
plotMap(MT, T=T, W=W2, P=[], text=f"seed=9999")
```



[61]:
```
P2 = createLoop(MT, [W2]+T, timing=True)
```

```
createTables:     0.40s
Solver:           0.23s    992 Constraints
Solver:           0.24s    997 Constraints
Solver:           0.54s  1,000 Constraints
Solver:           0.39s  1,001 Constraints
createLoop:       1.81s
```

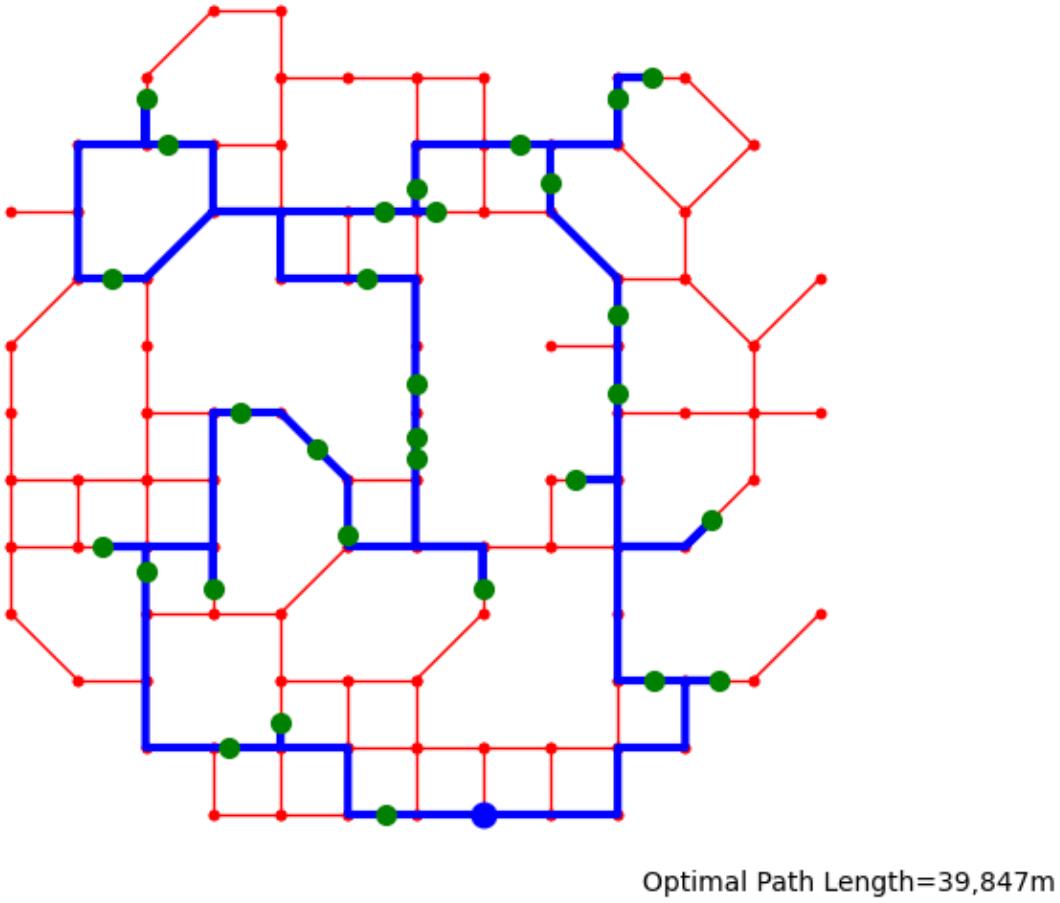
[62]:
```
PG2 = createLoopH(MT, [W2]+T, timing=True)
```

```
Floyd-Warshall:   0.24s
createLoopH:      0.26s
```

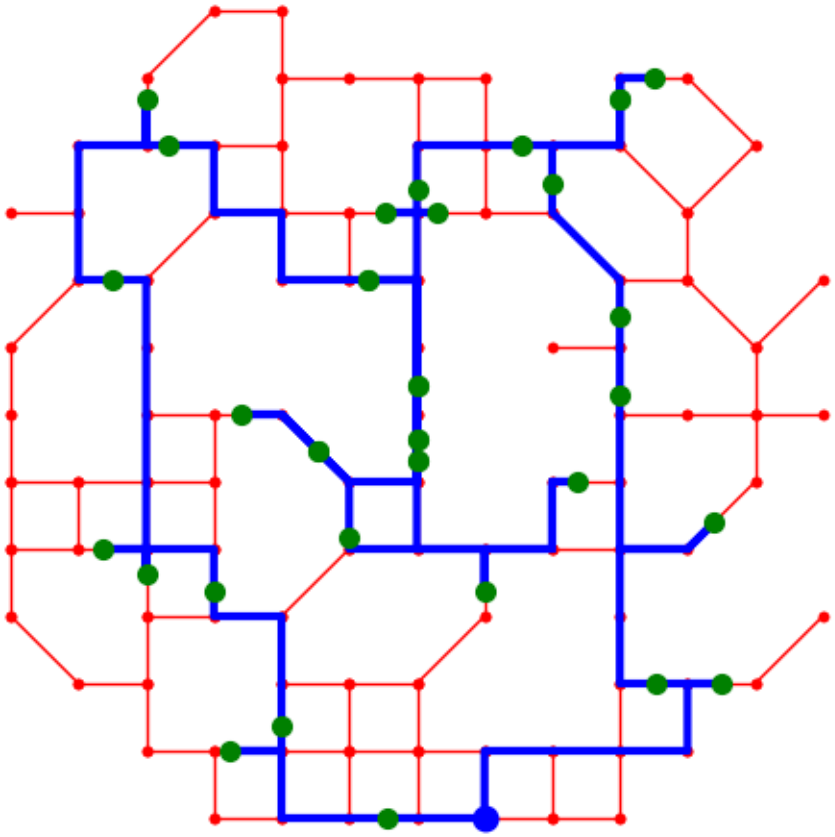
[63]:
```
PH2 = createLoopH(MT, [W2]+T, timing=True)
```

```
Floyd-Warshall:   0.18s
createLoopH:      0.20s
```

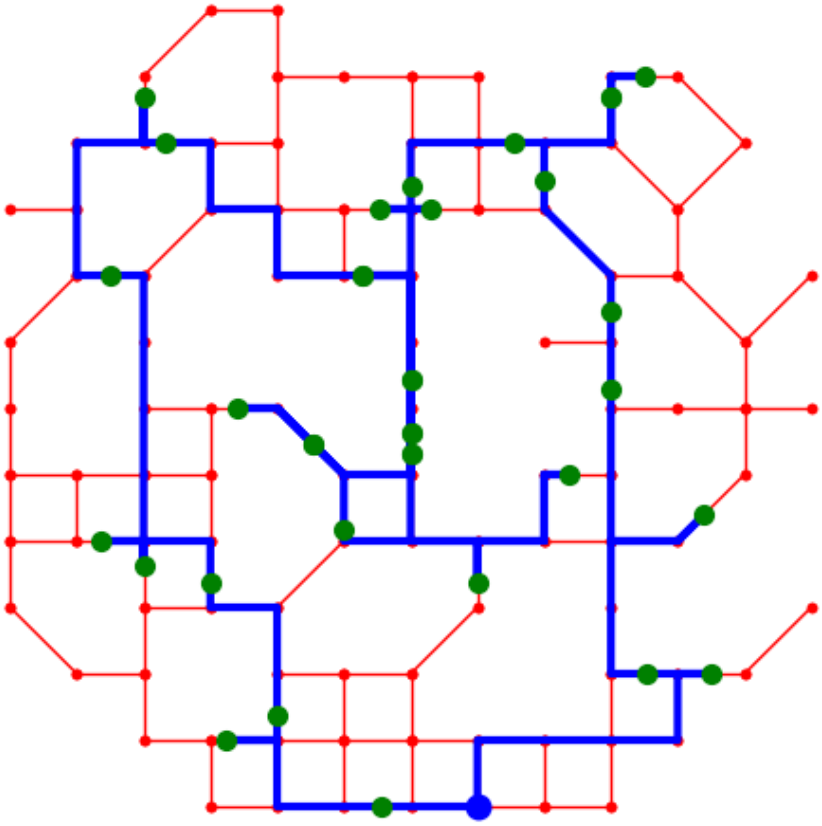
[64]:
```
plotMap(MT, T=T, W=W2, P=P2, text=f"Optimal Path Length={pathLength(P2):3,d}m")
```



[65]:
```
plotMap(MT, T=T, W=W2, P=PG2, text=f"Greedy Path Length={pathLength(PG2):3,d}m")
```



[66]:
```
plotMap(MT, T=T, W=W2, P=PH2, text=f"Heuristic Path Length={pathLength(PH2):
↪3,d}m")
```



5 Monte-Carlo Optimisation

This is an optimisation for the case of a fixed given number of customers to be served in one loop.

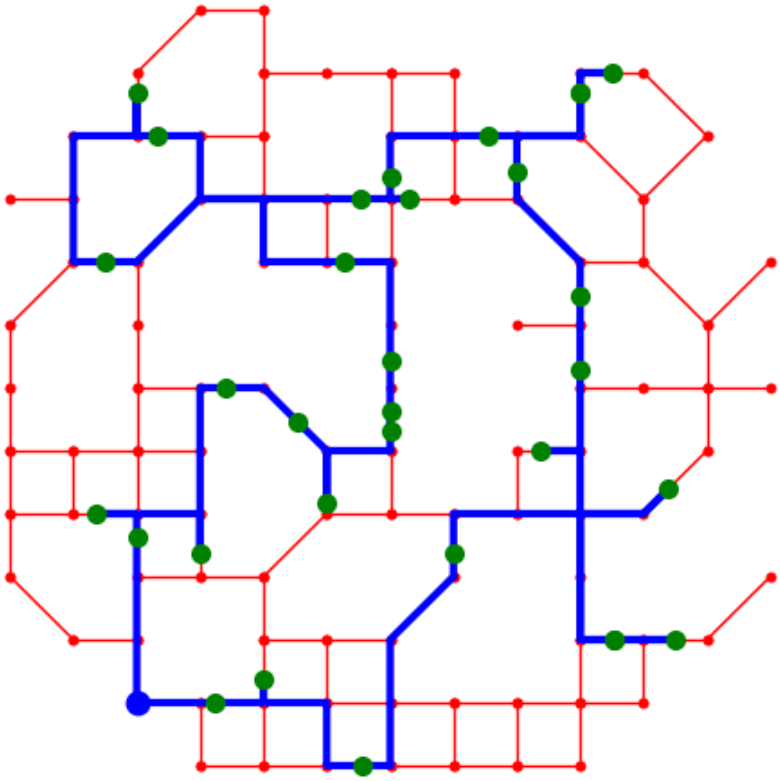
[67]:
```
def monte_carlo(M, T, k=math.inf, timing=False, plot=False):
    if timing:
        start_time = time.time()
    V, _ = M
    W = sorted(random.sample(V, k=min(len(V), k)))
    MT = addTargets(M, T)
    minL, minP, minW = math.inf, None, None
    for w in W:
        if minP is not None and w in minP:
            # any point on the current shortest loop will generat the same loop
            continue
        P = createLoop(MT, [w]+T)
        L = pathLength(P)
```

```
        if L<minL:
            minL, minP, minW = L, P, w
            print(f"pathlength: {L:6,d}m")
        if timing:
            print(f"     iteration:  {time.time()-start_time:6.2f}s")
    plotMap(MT, T=T, W=minW, P=minP, text=f"seed=9999 Path Length={minL:8.1f}m")
    return minW
```

[68]:
```
random.seed(0)
monte_carlo(M, T, timing=True, plot=True)
```

```
pathlength: 41,355m
     iteration:    1.91s
pathlength: 40,549m
     iteration:    3.30s
     iteration:    6.18s
     iteration:    8.47s
pathlength: 39,750m
     iteration:   10.00s
     iteration:   11.57s
     iteration:   14.64s
     iteration:   19.71s
     iteration:   21.90s
     iteration:   23.45s
     iteration:   27.42s
     iteration:   31.13s
     iteration:   33.16s
     iteration:   35.23s
     iteration:   39.31s
     iteration:   43.81s
     iteration:   51.82s
     iteration:   56.14s
     iteration:   58.04s
     iteration:   59.83s
     iteration:   62.87s
     iteration:   64.16s
     iteration:   67.17s
     iteration:   68.96s
     iteration:   70.38s
     iteration:   72.41s
     iteration:   75.36s
     iteration:   76.79s
     iteration:   78.04s
     iteration:   80.37s
     iteration:   82.26s
     iteration:   83.65s
     iteration:   84.96s
     iteration:   86.38s
```

```
iteration:   88.44s
iteration:   90.14s
iteration:   92.07s
iteration:   95.97s
iteration:   98.26s
iteration:  100.44s
iteration:  102.01s
iteration:  105.05s
iteration:  106.78s
iteration:  108.74s
iteration:  110.26s
iteration:  113.06s
```



seed=9999 Path Length= 39750.0m

[68]: (1760, 1200)

[ ]: