

Program 1 Preview: The Traveling Salesperson Problem

Dr. Demetrios Glinos
University of Central Florida

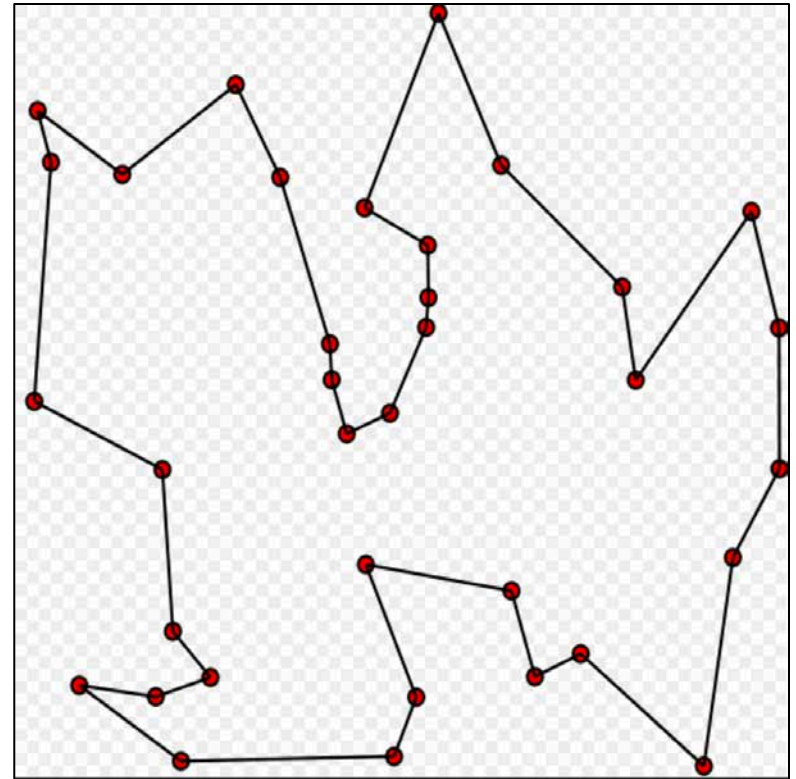
CAP4630 – Artificial Intelligence

Outline

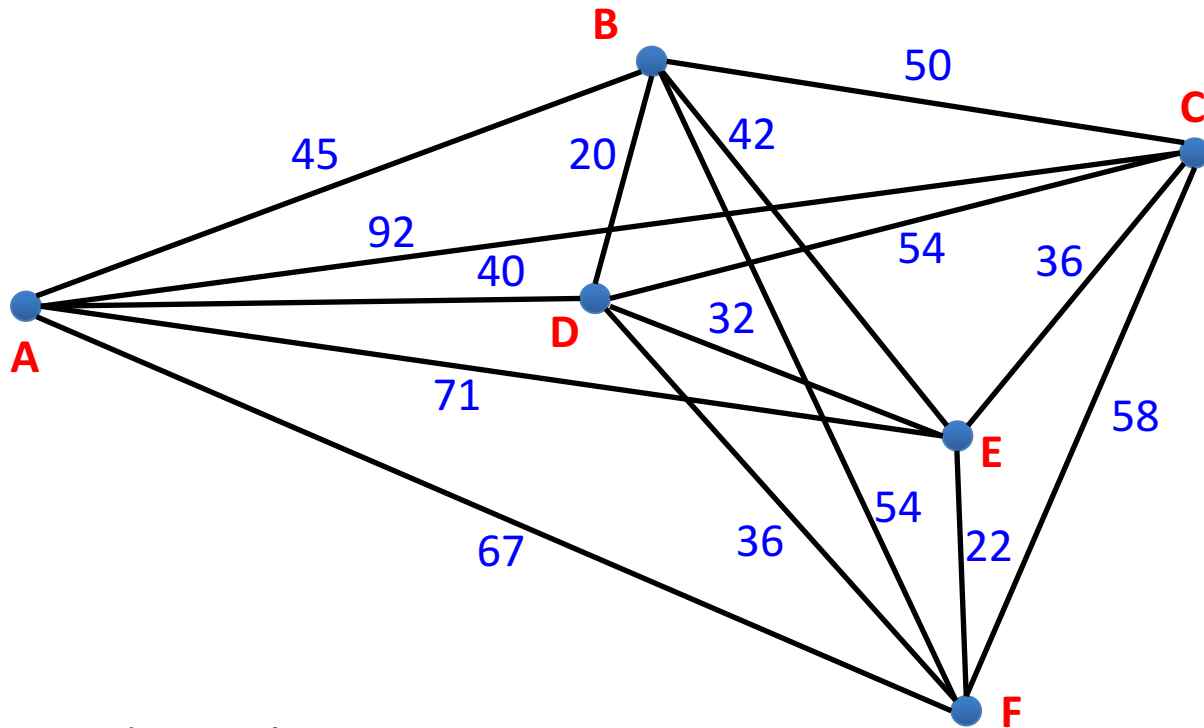
- Traveling Salesperson Problem
- Limits of Exhaustive Enumeration
- Applicability of Dijkstra's Algorithm
- Feasibility of Uniform Cost Search
- Nearest Neighbor Algorithm
- Repetitive Nearest Neighbor Algorithm
- Pac-Man's Tour
- Program 1

Traveling Salesperson Problem (TSP)

- **Given:**
 - n cities
 - distances or other “costs” between each pair of cities (n^2 entries)
- **Goal:**
 - Find the minimum cost path (**Hamiltonian circuit**) from one city that passes through each other city exactly once and returns to the starting city
 - Starting city can be fixed. **Why?**
- **Complexity**
 - Problem is known to be **NP-hard**



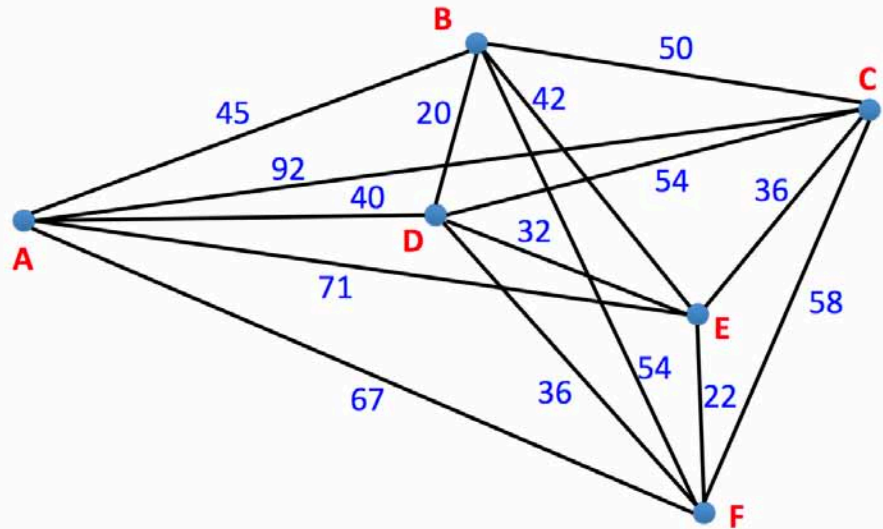
TSP Example



Nodes can be cities
Costs can be helicopter flight times

TSP Map and Cost Matrix

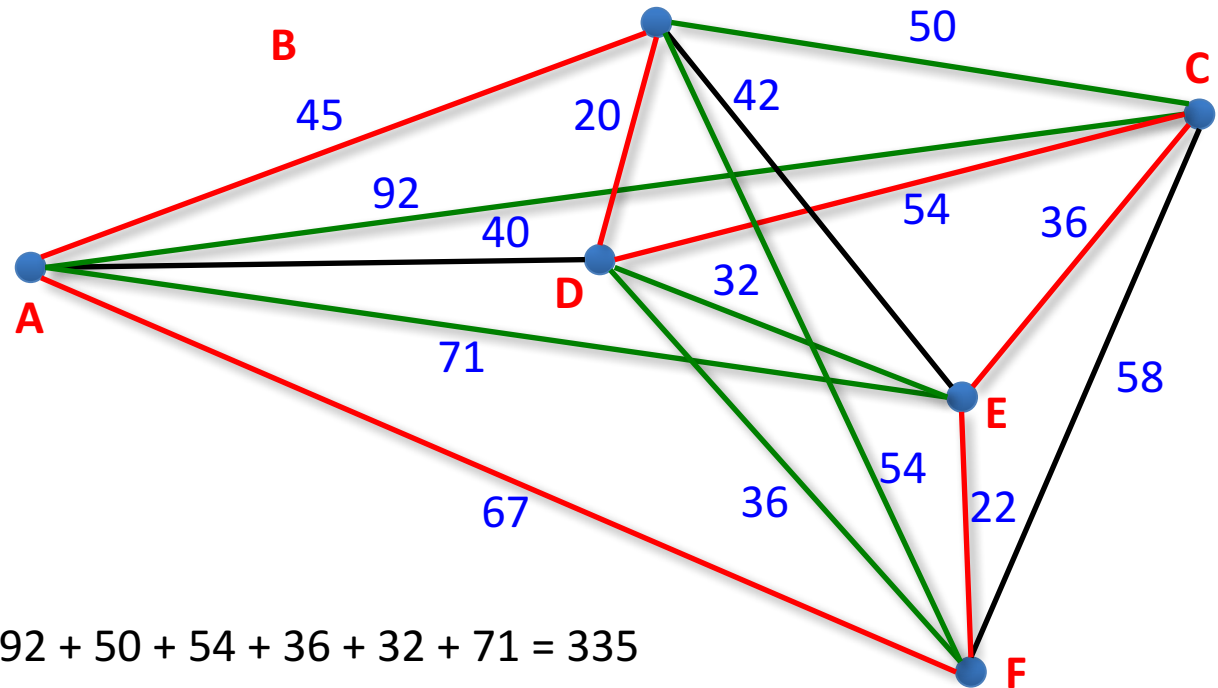
Note: This example shows a fully-connected graph



TSP Cost Matrix:

| From/to | A | B | C | D | E | F |
|---------|----|----|----|----|----|----|
| A | 0 | 45 | 92 | 40 | 71 | 67 |
| B | 45 | 0 | 50 | 20 | 42 | 54 |
| C | 92 | 50 | 0 | 54 | 36 | 58 |
| D | 40 | 20 | 54 | 0 | 32 | 36 |
| E | 71 | 42 | 36 | 32 | 0 | 22 |
| F | 67 | 54 | 58 | 36 | 22 | 0 |

TSP Circuits



Circuit **ACBFDEA** cost = $92 + 50 + 54 + 36 + 32 + 71 = 335$

Circuit **ABDCEFA** cost = $45 + 20 + 54 + 36 + 22 + 67 = 244$

...

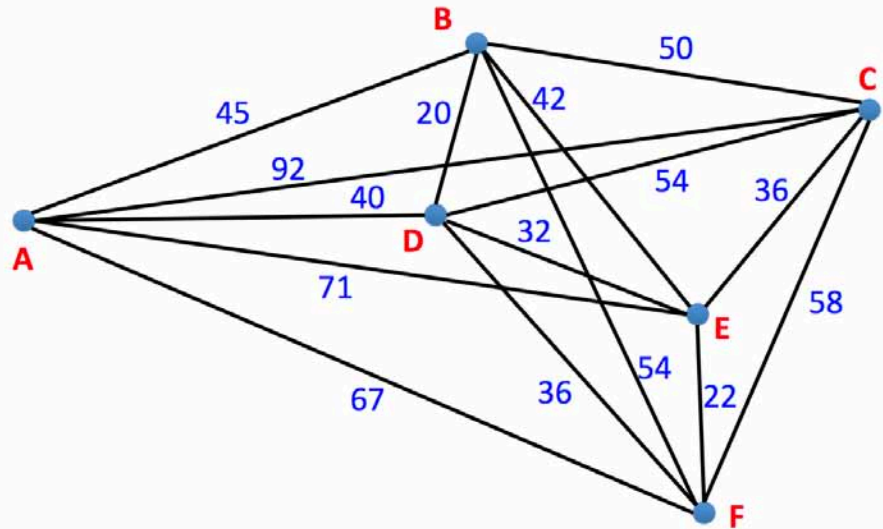
Circuit **ADBCEFA** cost = $40 + 20 + 50 + 36 + 22 + 67 = 235$

Circuit **ADFECBA** cost = $40 + 36 + 22 + 36 + 50 + 45 = 229$

The Limits of Exhaustive Enumeration

Q: How many total circuits are there?

A:



Exhaustive enumeration algorithm:

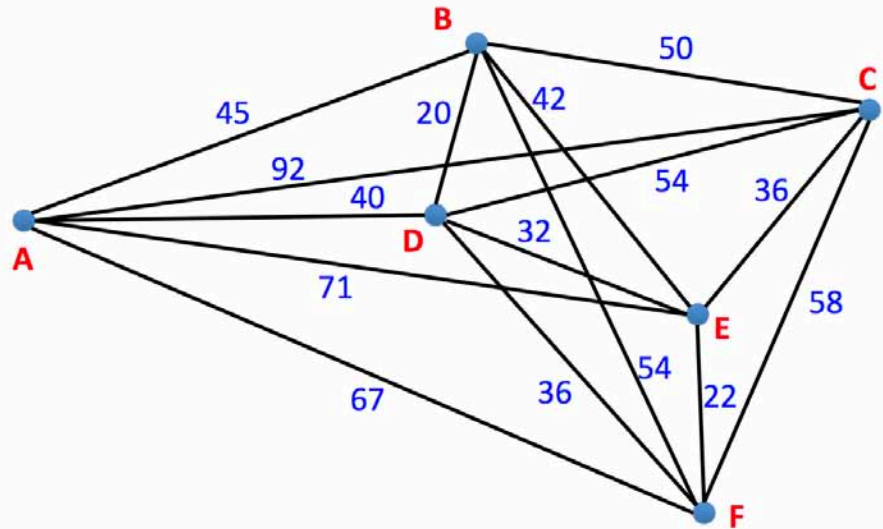
- compute the cost of each possible circuit
- choose a minimum cost circuit (there may be more than one)

The Limits of Exhaustive Enumeration

Q: How many total circuits are there?

A: $(n-1)!$, where $n = \#$ cities, if graph is fully-connected

If our computer can evaluate 1 million circuits per second:



Exhaustive enumeration algorithm:

- compute the cost of each possible circuit
- choose a minimum cost circuit (there may be more than one)

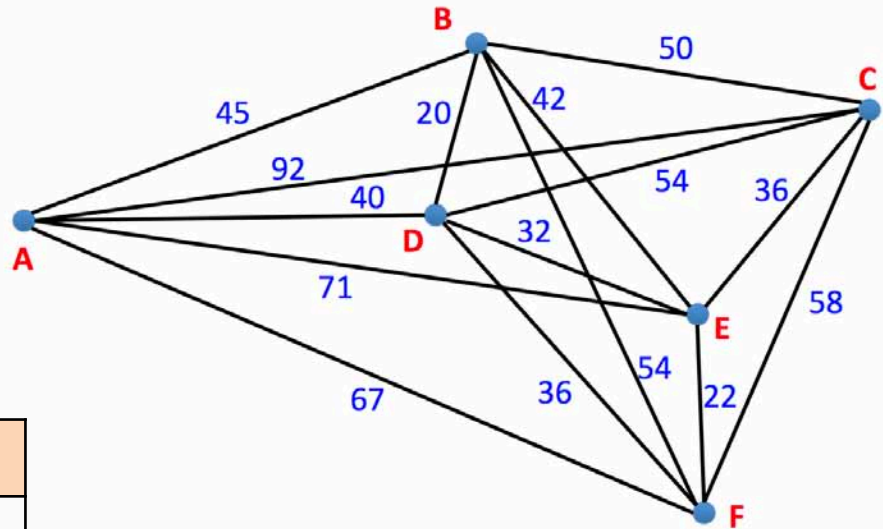
The Limits of Exhaustive Enumeration

Q: How many total circuits are there?

A: $(n-1)!$, where n = # cities, if graph is fully-connected

If our computer can evaluate 1 million circuits per second:

| N | Time to evaluate all circuits |
|------|-------------------------------|
| < 10 | instantaneous |
| 10 | about 4 seconds |
| 11 | about 40 seconds |
| 12 | about 8 minutes |
| 13 | nearly 2 hours |
| 14 | a little over a day |
| 15 | about a year |
| 20 | over a million years |



Exhaustive enumeration algorithm:

- compute the cost of each possible circuit
- choose a minimum cost circuit (there may be more than one)

Applicability of Dijkstra's Algorithm

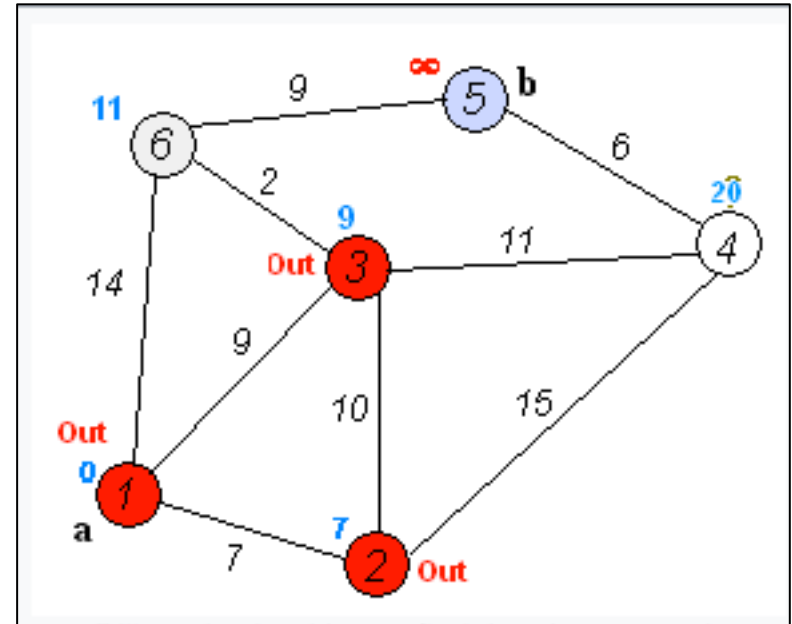
- Dijkstra's Algorithm

- finds min cost paths
- worst-case time complexity is

$$O(E + |V|\log|V|)$$

- Unfortunately, does not solve TSP because:

- nodes are examined in order of current min cost *from the source node*
- as a result
 - successive node expansions are not necessarily directly connected
 - so, sequence of Dijkstra expansions cannot be used to determine circuit
- output of Dijkstra's algorithm is one row of TSP cost matrix
 - using each node as the source give is the entire cost matrix
 - the cost matrix is only the *input* to the TSP problem
 - our Pac-Man library contains utility methods to calculate these values



Source: Wikipedia

Feasibility of Uniform Cost Search

- UCS space and time complexity is $O(b^{C^*/\epsilon})$
 - exponential in “effective depth”
 - where C^* = cost of solution
 - b is average branching factor
 - and ϵ = minimum arc cost
- For a 20-node TSP problem on a Pac-Man maze
 - $C^* \approx 230$
 - $\epsilon = 1$
 - $b \approx 1.148$
 - Number of nodes examined $\approx 61,183,491,137,155$
 - If can examine 1 million nodes per second, this is ≈ 708 days (1.94 years)
 - Actual examination rate is much slower
- Result: UCS not practical for large TSPs, either

What does C^/ϵ represent?*

What happens to the fringe and explored set?

Nearest Neighbor Algorithm (NNA)

- A heuristic (not optimal), but generally gives good results
- Basic idea
 - For n nodes, there are n choices to make
 - From the start node, choose the nearest unvisited neighbor as the next node
 - Repeat until done

- Starting from A on this graph, path is:

[{A}, 0]

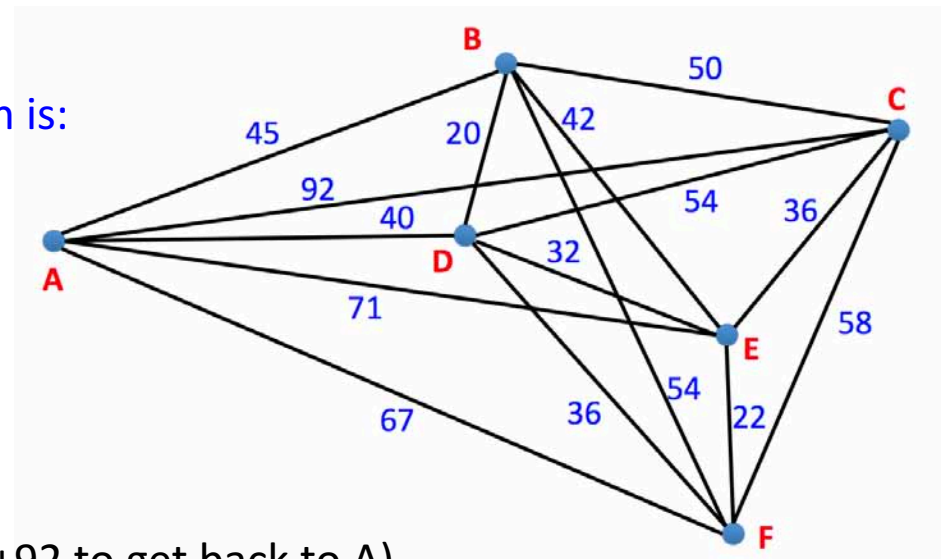
[{A, D}, 40]

[{A, D, B}, 60]

[{A, D, B, E}, 102]

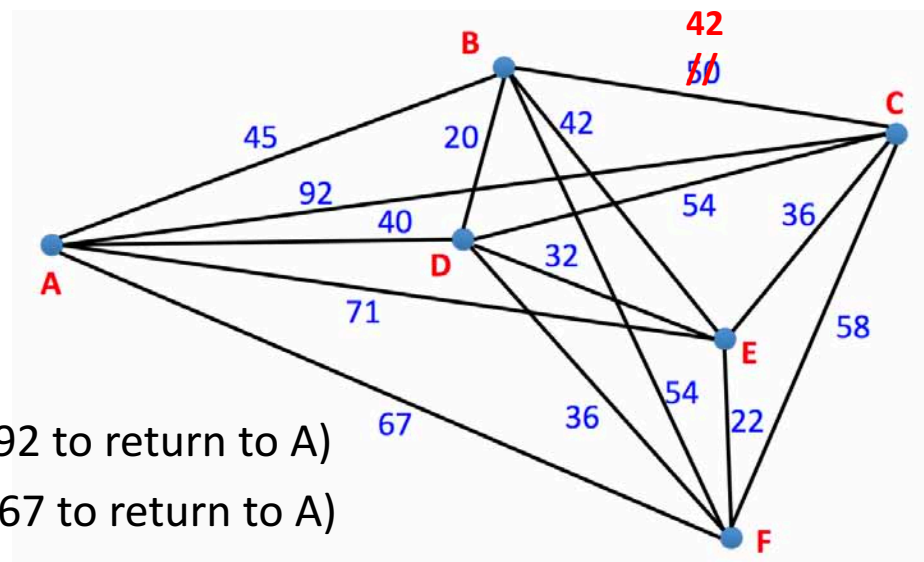
[{A, D, B, E, F}, 124]

[{A, D, B, E, F, C}, 182] (+92 to get back to A)



Multiple Partial Plans

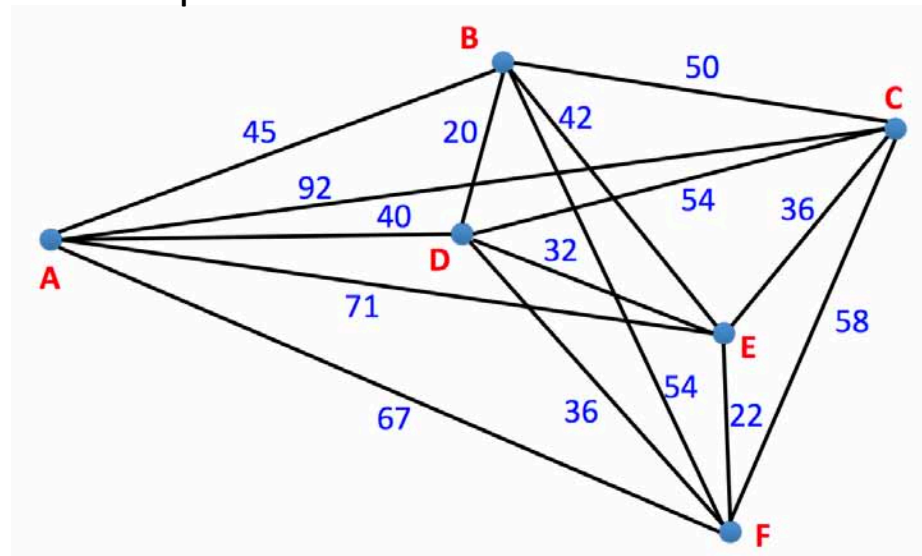
- Basic NNA keeps just 1 partial plan and builds on it
- When multiple neighbors are equally close, we must split into additional partial plans
- Example: Suppose path from B to C were 42 (same as from B to E):



- Start with 1 plan
 - get [{A, D, B}, 60]
- Now split into 2 plans:
 - Plan 1: [{A, D, B, E, F, C}, 182] (+92 to return to A)
 - Plan 2: [{A, D, B, C, E, F}, 160] (+67 to return to A)
 - Choose Plan 2 as the better plan

Repetitive Nearest Neighbor Algorithm (RNNA)

- Basic idea:
 - Apply NNA using each vertex as the starting node
 - For n vertices, there will be at least n complete paths to compare (more if need to split paths as on previous slide)
 - Choose the lowest cost path



Repetitive Nearest Neighbor Algorithm (RNNA)

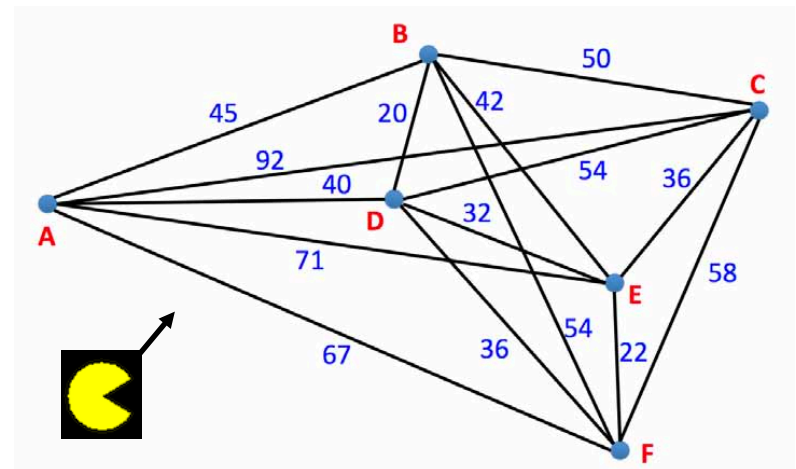
- Now, if we have an additional starting node (e.g., Pac-Man's starting position):
 - For **complete (Hamiltonian) circuits**, we know we can start anywhere and reach all other nodes
 - E.g., **A B D C E F A** is same as **C E F A B D C**, etc.
 - So, shortest path with additional start node is: shortest circuit + shortest link from start node to a node in the path
 - But if paths are **incomplete circuits**, as with Pac-Man, we must compare path costs from start node to start of each path + cost of each path
- Thus, we compare costs of:

(Pac-Man to A) + (NNA from A)

(Pac-Man to B) + (NNA from B)

...

(Pac-Man to F) + (NNA from F)

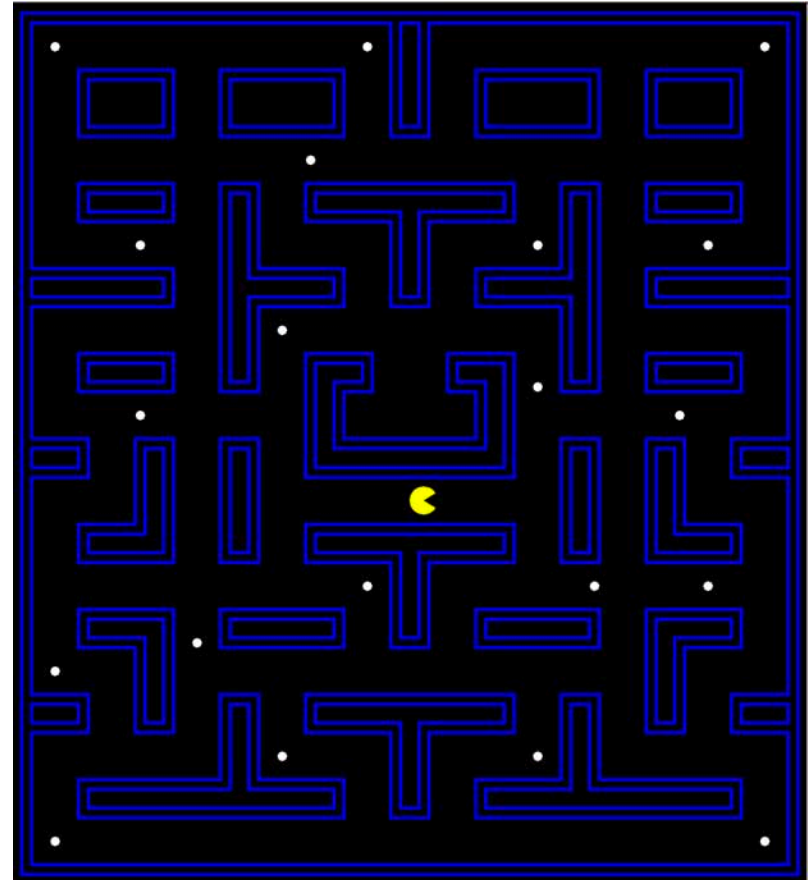


Pac-Man's Tour

- Problem is basic TSP, except we do not return to the start location
- 20 food pellets + Pac-Man's starting location
 - 20! total tours (fully-connected)
- Brute force and UCS infeasible

| Algorithm | Path length |
|-----------|-------------|
| Replan | 246 |
| NNA | 236 |
| RNNA | 217 |

- Task is to implement RNNA for this maze
- Full details are on Webcourses

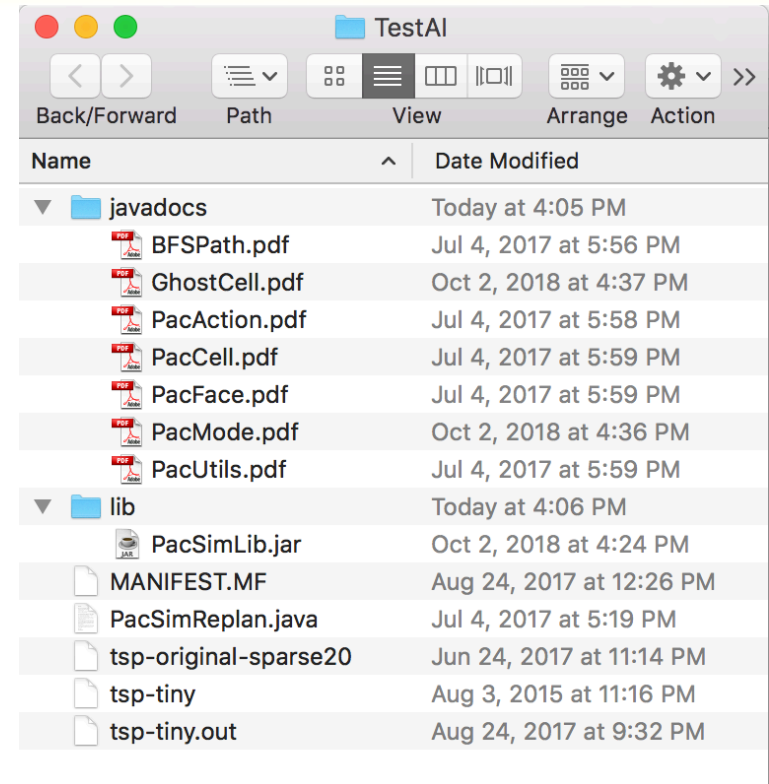


demos: [tspreplan](#), [tspnna](#), [tsprnna](#)

Creating an Executable Jar for the Sample Agent

Step 1: Download the TestAI.zip file and unzip it to your desktop

For Windows systems, replace the colon with a semicolon



Step 2: Open a command window, navigate to the TestAI folder, and run these commands:

Compile command: `javac -cp lib/PacSimLib.jar PacSimReplan.java`

Run command: `java -cp .:lib/PacSimLib.jar PacSimReplan tsp-tiny`

Creating an Executable Jar for the Sample Agent

Step 3: Create an executable JAR file that contains both source and class files:

```
jar cfm PacSimReplan.jar MANIFEST.MF *.java *.class
```

Note: This instruction uses the MANIFEST.MF manifest file that is included in the distribution

Step 4: Test the JAR file by running it:

```
java -jar PacSimReplan.jar tsp-tiny
```

Step 5: Check to be sure that the JAR file contains the source files, too:

```
jar -tvf PacSimReplan.jar
```

Submitting an Executable Jar for Your Agent

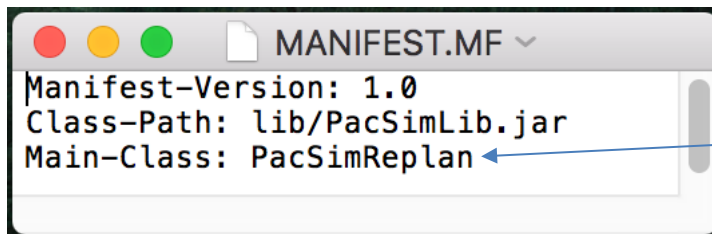
(These instructions follow the sample procedure and assume that your agent is in the file PacSimRNNA.java)

Copy your source file to the TestAI folder, and then execute:

Compile command: `javac -cp lib/PacSimLib.jar PacSimRNNA.java`

Run command: `java -cp ./lib/PacSimLib.jar PacSimRNNA tsp-tiny`

Modify the MANIFEST.MF file to name the Java class for your agent



change this to PacSimRNNA

Execute these commands:

`jar cfm PacSimRNNA.jar MANIFEST.MF *.java *.class` ← create the jar

`jar -tvf PacSimRNNA.jar` ← make sure it includes source files

`java -jar PacSimRNNA.jar tsp-tiny` ← test the jar

If everything works as expected, submit the JAR file on Webcourses

PacSimReplan.java (1)

```
1
2  import java.awt.Point;
3  import java.util.ArrayList;
4  import java.util.List;
5  import pacsim.BFSPath;
6  import pacsim.PacAction;
7  import pacsim.PacCell;
8  import pacsim.PacFace;
9  import pacsim.PacSim;
10 import pacsim.PacUtils;
11 import pacsim.PacmanCell;
12
13 /**
14  * Simple Re-planning Search Agent
15  * @author Dr. Demetrios Glinos
16  */
17 public class PacSimReplan implements PacAction {
18
19     private List<Point> path;
20     private int simTime;
21
22     public PacSimReplan( String fname ) {
23         PacSim sim = new PacSim( fname );
24         sim.init(this);
25     }
26
27     public static void main( String[] args ) {
28         System.out.println("\nTSP using simple replanning agent by Dr. Demetrios Glinos:");
29         System.out.println("\nMaze : " + args[ 0 ] + "\n" );
30         new PacSimReplan( args[ 0 ] );
31     }
32
33     @Override
34     public void init() {
35         simTime = 0;
36         path = new ArrayList();
37     }
38 }
```

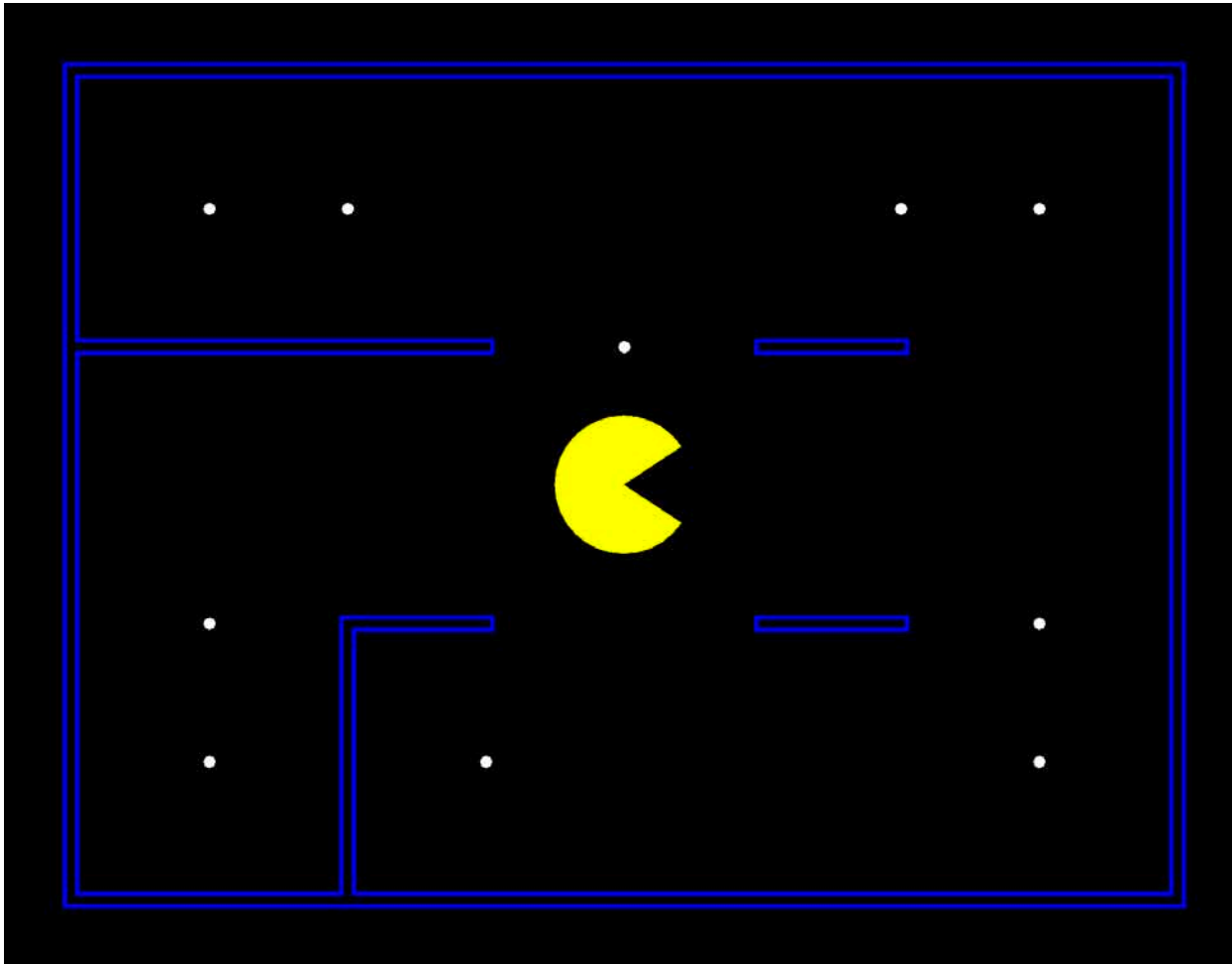
PacSimReplan.java (2)

```

38
39 @Override
40 public PacFace action( Object state ) {
41
42     PacCell[][] grid = (PacCell[][]) state;
43     PacmanCell pc = PacUtils.findPacman( grid );
44
45     // make sure Pac-Man is in this game
46     if( pc == null ) return null;
47
48     // if current path completed (or just starting out),
49     // select a the nearest food using the city-block
50     // measure and generate a path to that target
51
52     if( path.isEmpty() ) {
53         Point tgt = PacUtils.nearestFood( pc.getLoc(), grid);
54         path = BFSPath.getPath(grid, pc.getLoc(), tgt);
55
56         System.out.println("Pac-Man currently at: [ " + pc.getLoc().x
57             + ", " + pc.getLoc().y + " ]");
58         System.out.println("Setting new target : [ " + tgt.x
59             + ", " + tgt.y + " ]");
60     }
61
62     // take the next step on the current path
63
64     Point next = path.remove( 0 );
65     PacFace face = PacUtils.direction( pc.getLoc(), next );
66     System.out.printf( "%5d : From [ %2d, %2d ] go %s%n",
67         ++simTime, pc.getLoc().x, pc.getLoc().y, face );
68     return face;
69 }
70 }

```

Development maze: *tsp-tiny*



Required Output Format (1)

1

TSP using Repetitive Nearest Neighbor Algorithm by Dr. Demetrios Glinos:

Maze : tsp-tiny

2

Cost table:

| | | | | | | | | | | |
|---|----|---|----|---|---|---|---|----|---|----|
| 0 | 5 | 4 | 5 | 4 | 3 | 1 | 4 | 5 | 4 | 5 |
| 5 | 0 | 9 | 10 | 1 | 8 | 4 | 5 | 6 | 9 | 10 |
| 4 | 9 | 0 | 1 | 8 | 7 | 5 | 8 | 9 | 8 | 9 |
| 5 | 10 | 1 | 0 | 9 | 8 | 6 | 9 | 10 | 9 | 10 |
| 4 | 1 | 8 | 9 | 0 | 7 | 3 | 4 | 5 | 8 | 9 |
| 3 | 8 | 7 | 8 | 7 | 0 | 4 | 7 | 8 | 5 | 4 |
| 1 | 4 | 5 | 6 | 3 | 4 | 0 | 3 | 4 | 5 | 6 |
| 4 | 5 | 8 | 9 | 4 | 7 | 3 | 0 | 1 | 4 | 5 |
| 5 | 6 | 9 | 10 | 5 | 8 | 4 | 1 | 0 | 3 | 4 |
| 4 | 9 | 8 | 9 | 8 | 5 | 5 | 4 | 3 | 0 | 1 |
| 5 | 10 | 9 | 10 | 9 | 4 | 6 | 5 | 4 | 1 | 0 |

3

Food Array:

0 : (1,1)
 1 : (1,4)
 2 : (1,5)
 3 : (2,1)
 4 : (3,5)
 5 : (4,2)
 6 : (6,1)
 7 : (7,1)
 8 : (7,4)
 9 : (7,5)

Required Output Format (2)

4

Population at step 1 :

```
0 : cost=1 : [(4,2),1]
1 : cost=3 : [(3,5),3]
2 : cost=4 : [(1,4),4]
3 : cost=4 : [(2,1),4]
4 : cost=4 : [(6,1),4]
5 : cost=4 : [(7,4),4]
6 : cost=5 : [(1,1),5]
7 : cost=5 : [(1,5),5]
8 : cost=5 : [(7,1),5]
9 : cost=5 : [(7,5),5]
```

For RNA, there is an initial partial path from Pac-Man's initial location to each food dot

Note branching for equidistant successor nodes

Population at step 10 :

```
0 : cost=27 : [(4,2),1][(2,1),3][(1,1),1][(6,1),5][(7,1),1][(7,4),3][(7,5),1][(3,5),4][(1,4),7][(1,5),1]
1 : cost=29 : [(1,4),4][(1,5),1][(4,2),6][(2,1),3][(1,1),1][(6,1),5][(7,1),1][(7,4),3][(7,5),1][(3,5),4]
2 : cost=29 : [(1,5),5][(1,4),1][(4,2),5][(2,1),3][(1,1),1][(6,1),5][(7,1),1][(7,4),3][(7,5),1][(3,5),4]
3 : cost=29 : [(3,5),3][(7,5),4][(7,4),1][(7,1),3][(6,1),1][(4,2),3][(2,1),3][(1,1),1][(1,4),9][(1,5),1]
4 : cost=29 : [(2,1),4][(1,1),1][(4,2),4][(6,1),3][(7,1),1][(7,4),3][(7,5),1][(3,5),4][(1,4),7][(1,5),1]
5 : cost=29 : [(1,1),5][(2,1),1][(4,2),3][(6,1),3][(7,1),1][(7,4),3][(7,5),1][(3,5),4][(1,4),7][(1,5),1]
6 : cost=31 : [(4,2),1][(6,1),3][(7,1),1][(7,4),3][(7,5),1][(3,5),4][(1,4),7][(1,5),1][(2,1),9][(1,1),1]
7 : cost=31 : [(4,2),1][(6,1),3][(7,1),1][(7,4),3][(7,5),1][(3,5),4][(1,4),7][(1,5),1][(2,1),9][(1,1),1]
8 : cost=31 : [(6,1),4][(7,1),1][(7,4),3][(7,5),1][(3,5),4][(4,2),4][(2,1),3][(1,1),1][(1,4),9][(1,5),1]
9 : cost=31 : [(3,5),3][(4,2),4][(2,1),3][(1,1),1][(6,1),5][(7,1),1][(7,4),3][(7,5),1][(1,4),9][(1,5),1]
10 : cost=31 : [(1,4),4][(1,5),1][(4,2),6][(6,1),3][(7,1),1][(7,4),3][(7,5),1][(3,5),4][(2,1),7][(1,1),1]
11 : cost=31 : [(1,5),5][(1,4),1][(4,2),5][(6,1),3][(7,1),1][(7,4),3][(7,5),1][(3,5),4][(2,1),7][(1,1),1]
12 : cost=33 : [(7,4),4][(7,5),1][(3,5),4][(4,2),4][(2,1),3][(1,1),1][(6,1),5][(7,1),1][(1,4),9][(1,5),1]
13 : cost=33 : [(7,4),4][(7,5),1][(3,5),4][(4,2),4][(6,1),3][(7,1),1][(2,1),5][(1,1),1][(1,4),9][(1,5),1]
14 : cost=33 : [(7,4),4][(7,5),1][(7,1),4][(6,1),1][(4,2),3][(2,1),3][(1,1),1][(3,5),8][(1,4),7][(1,5),1]
15 : cost=33 : [(7,5),5][(7,4),1][(7,1),3][(6,1),1][(4,2),3][(2,1),3][(1,1),1][(3,5),8][(1,4),7][(1,5),1]
16 : cost=35 : [(3,5),3][(4,2),4][(6,1),3][(7,1),1][(7,4),3][(7,5),1][(1,4),9][(1,5),1][(2,1),9][(1,1),1]
17 : cost=35 : [(3,5),3][(4,2),4][(6,1),3][(7,1),1][(7,4),3][(7,5),1][(2,1),9][(1,1),1][(1,4),9][(1,5),1]
18 : cost=35 : [(7,1),5][(6,1),1][(4,2),3][(2,1),3][(1,1),1][(3,5),8][(7,5),4][(7,4),1][(1,4),8][(1,5),1]
```

Note: There will be one population step for each food dot in the tour, but you should just report the first and last steps

Required Output Format (3)

5

Time to generate plan: 27 msec

6

Solution moves:

7

```

1 : From [ 4, 3 ] go N
2 : From [ 4, 2 ] go N
3 : From [ 4, 1 ] go W
4 : From [ 3, 1 ] go W
5 : From [ 2, 1 ] go W
6 : From [ 1, 1 ] go E
7 : From [ 2, 1 ] go E
8 : From [ 3, 1 ] go E
9 : From [ 4, 1 ] go E
10 : From [ 5, 1 ] go E
11 : From [ 6, 1 ] go E
12 : From [ 7, 1 ] go S
13 : From [ 7, 2 ] go S
14 : From [ 7, 3 ] go S
15 : From [ 7, 4 ] go S
16 : From [ 7, 5 ] go W
17 : From [ 6, 5 ] go W
18 : From [ 5, 5 ] go W
19 : From [ 4, 5 ] go W
20 : From [ 3, 5 ] go E
21 : From [ 4, 5 ] go N
22 : From [ 4, 4 ] go N
23 : From [ 4, 3 ] go W
24 : From [ 3, 3 ] go W
25 : From [ 2, 3 ] go W
26 : From [ 1, 3 ] go S
27 : From [ 1, 4 ] go S

```

8

Game over: Pacman has eaten all the food

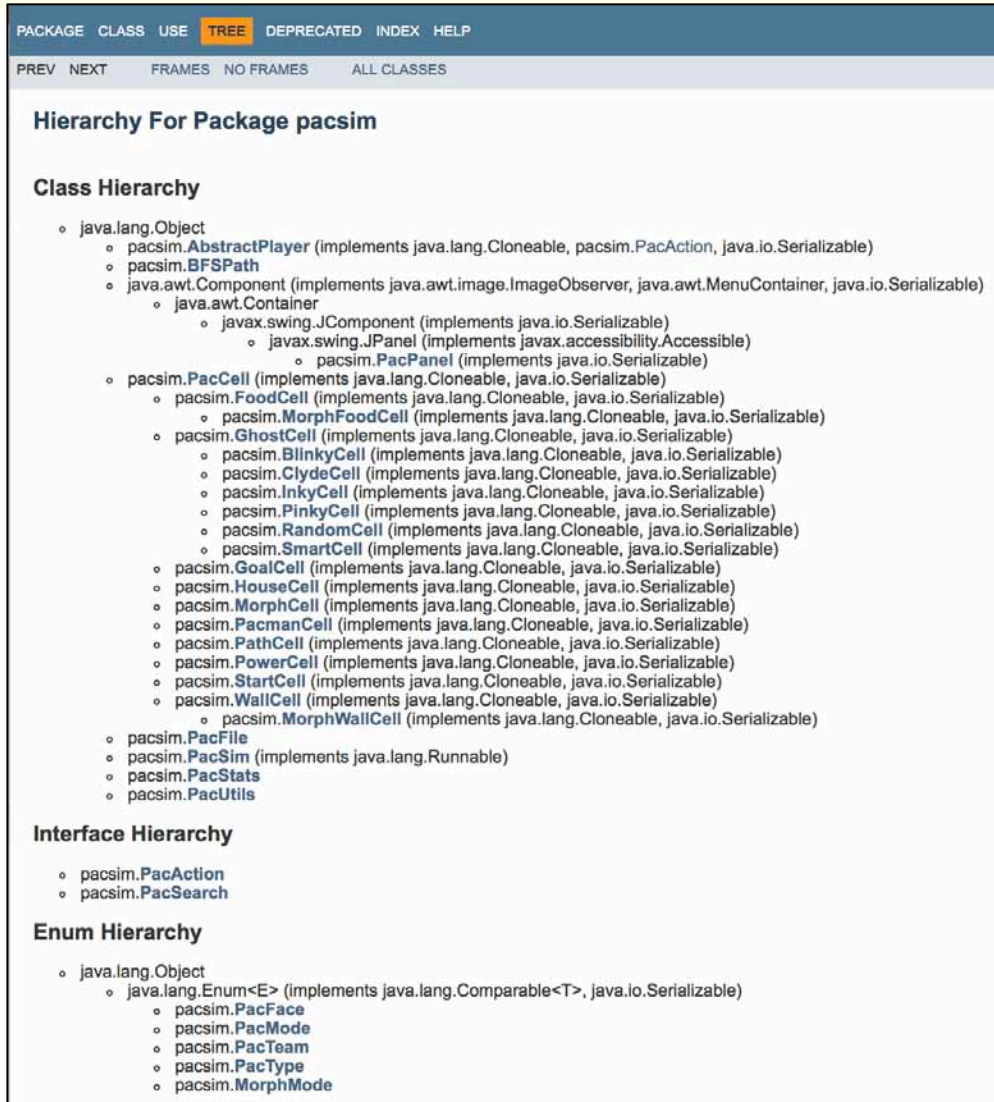
*This statement is generated
by the simulation engine
(not by the agent program)*

Use the Javadocs

Javadocs (in PDF form) are provided for what you will need:

- BFSPATH class
- PacAction interface
- PacCell class
- PacFace enum
- PacUtils class
- plus others that you will need for other assignments

NOTE: Although PacUtils contains many utility methods, you will need to create your own methods, comparators, and/or classes to implement RNA



PACKAGE CLASS USE **TREE** DEPRECATED INDEX HELP

PREV NEXT FRAMES NO FRAMES ALL CLASSES

Hierarchy For Package pacsim

Class Hierarchy

- java.lang.Object
 - pacsim.AbstractPlayer (implements java.lang.Cloneable, pacsim.PacAction, java.io.Serializable)
 - pacsim.BFSPATH
 - java.awt.Component (implements java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable)
 - java.awt.Container
 - javax.swing.JComponent (implements java.io.Serializable)
 - javax.swing.JPanel (implements javax.accessibility.Accessible)
 - pacsim.PacPanel (implements java.io.Serializable)
 - pacsim.PacCell (implements java.lang.Cloneable, java.io.Serializable)
 - pacsim.FoodCell (implements java.lang.Cloneable, java.io.Serializable)
 - pacsim.MorphFoodCell (implements java.lang.Cloneable, java.io.Serializable)
 - pacsim.GhostCell (implements java.lang.Cloneable, java.io.Serializable)
 - pacsim.BlinkyCell (implements java.lang.Cloneable, java.io.Serializable)
 - pacsim.ClydeCell (implements java.lang.Cloneable, java.io.Serializable)
 - pacsim.InkyCell (implements java.lang.Cloneable, java.io.Serializable)
 - pacsim.PinkyCell (implements java.lang.Cloneable, java.io.Serializable)
 - pacsim.RandomCell (implements java.lang.Cloneable, java.io.Serializable)
 - pacsim.SmartCell (implements java.lang.Cloneable, java.io.Serializable)
 - pacsim.GoalCell (implements java.lang.Cloneable, java.io.Serializable)
 - pacsim.HouseCell (implements java.lang.Cloneable, java.io.Serializable)
 - pacsim.MorphCell (implements java.lang.Cloneable, java.io.Serializable)
 - pacsim.PacmanCell (implements java.lang.Cloneable, java.io.Serializable)
 - pacsim.PathCell (implements java.lang.Cloneable, java.io.Serializable)
 - pacsim.PowerCell (implements java.lang.Cloneable, java.io.Serializable)
 - pacsim.StartCell (implements java.lang.Cloneable, java.io.Serializable)
 - pacsim.WallCell (implements java.lang.Cloneable, java.io.Serializable)
 - pacsim.MorphWallCell (implements java.lang.Cloneable, java.io.Serializable)
 - pacsim.PacFile
 - pacsim.PacSim (implements java.lang.Runnable)
 - pacsim.PacStats
 - pacsim.PacUtils

Interface Hierarchy

- pacsim.PacAction
- pacsim.PacSearch

Enum Hierarchy

- java.lang.Enum
 - java.lang.Enum<E> (implements java.lang.Comparable<T>, java.io.Serializable)
 - pacsim.PacFace
 - pacsim.PacMode
 - pacsim.PacTeam
 - pacsim.PacType
 - pacsim.MorphMode