Program 1 Desiderata

Dr. Demetrios Glinos
University of Central Florida

CAP4630 – Artificial Intelligence

Fall 2015

de·sid·er·a·tum

/diˌsidəˈrätəm,-ˈrātəm,-ˌzidə-/

noun
plural noun: **desiderata**

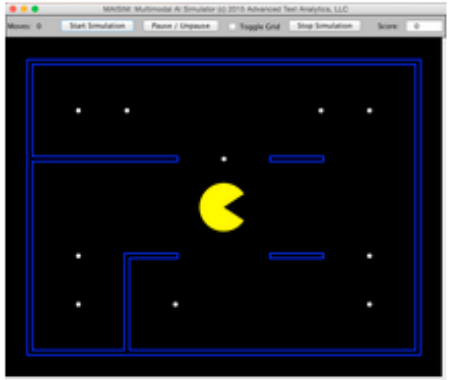
something that is needed or wanted.
"integrity was a desideratum"
synonyms: requirement, prerequisite, need, indispensable thing, sine qua non, essential, requisite, necessary
"integrity was a desideratum"

Topics

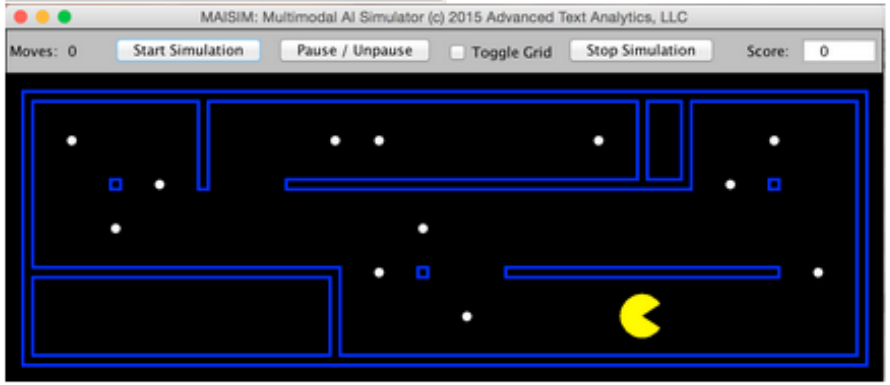
- Applicability of Dijkstra's algorithm
- How to control Pac-Man

Topic 1: Applicability of Dijkstra's algorithm

The Assignment



10 dots



12 dots

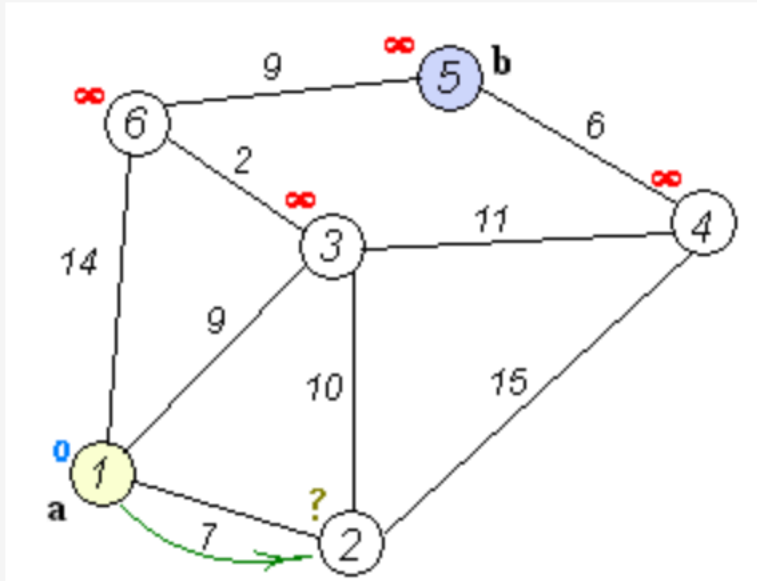
- Implement UCS
  - Use graph search
  - Goal test: all dots eaten
  - Find optimal solutions for both given maps
- Drive Pac-Man through solution
- This is a traveling salesman problem

Dijkstra's Algorithm

from the Wikipedia page:

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.[1][2] The algorithm exists in many variants; Dijkstra's original variant found the shortest path between two nodes,[2] but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest path tree.

The algorithm runs in polynomial time



Using Food Dots as the Nodes

Tempting approach:

1. calculate distances from start to each dot
2. take shortest path to first dot
3. now, calculate distances from first dot to remaining unvisited dots
4. take shortest path to next dot
5. repeat steps 3 and 4 until all dots eaten

Using Food Dots as the Nodes

Tempting approach:

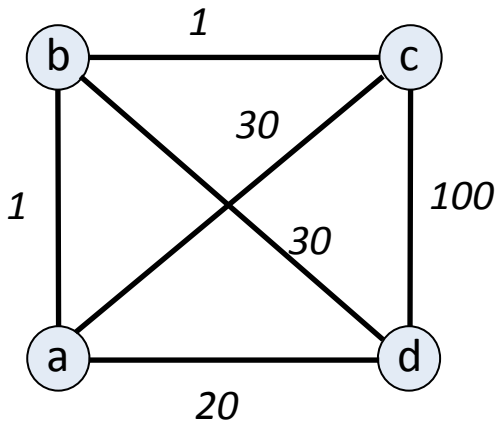
1. calculate distances from start to each dot
2. take shortest path to first dot
3. now, calculate distances from first dot to remaining unvisited dots
4. take shortest path to next dot
5. repeat steps 3 and 4 until all dots eaten

Q: Is optimality guaranteed?

Using Food Dots as the Nodes

Tempting approach:

1. calculate distances from start to each dot
2. take shortest path to first dot
3. now, calculate distances from first dot to remaining unvisited dots
4. take shortest path to next dot
5. repeat steps 3 and 4 until all dots eaten



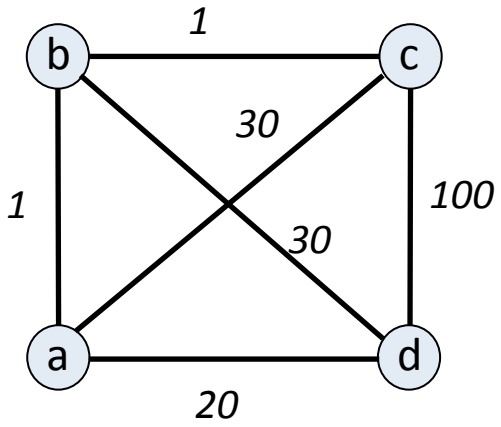
Q: Is optimality guaranteed?

Using Food Dots as the Nodes

Tempting approach:

1. calculate distances from start to each dot
2. take shortest path to first dot
3. now, calculate distances from first dot to remaining unvisited dots
4. take shortest path to next dot
5. repeat steps 3 and 4 until all dots eaten

Q: Is optimality guaranteed? **No**



Dijkstra path: abcd, cost=102

optimal path: acbd, cost=61

Naive Brute Force Search

Basic idea:

1. Enumerate all possible orderings
2. Compute cost (path length) for each
3. Choose the lowest cost path

Naive Brute Force Search

Basic idea:

1. Enumerate all possible orderings
2. Compute cost (path length) for each
3. Choose the lowest cost path

We must consider n! choices:

10! = 3,628,800

12! = 479,001,600

Naive Brute Force Search

Basic idea:

1. Enumerate all possible orderings
2. Compute cost (path length) for each
3. Choose the lowest cost path

We must consider n! choices:

10! = 3,628,800

12! = 479,001,600

Q: How do we calculate the distances from point to point, anyway?

Naive Brute Force Search

Basic idea:

1. Enumerate all possible orderings
2. Compute cost (path length) for each
3. Choose the lowest cost path

We must consider n! choices:

10! = 3,628,800

12! = 479,001,600

Q: How do we calculate the distances from point to point, anyway?

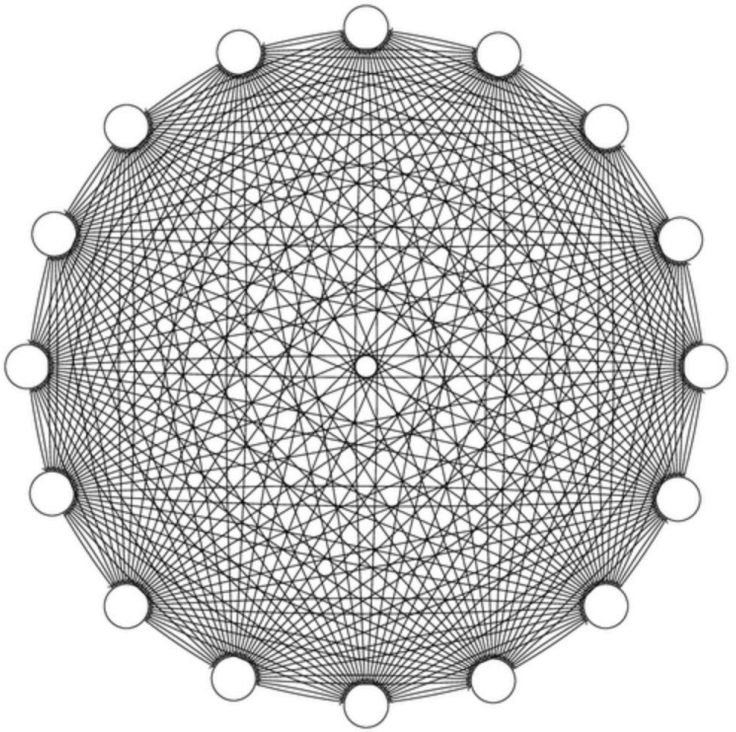
BFS, UCS, or A* (take your pick)

What our TSP graph really looks like

- If we can't know distances between nodes without BFS/UCS/A*, then every spot that is not a wall is a node

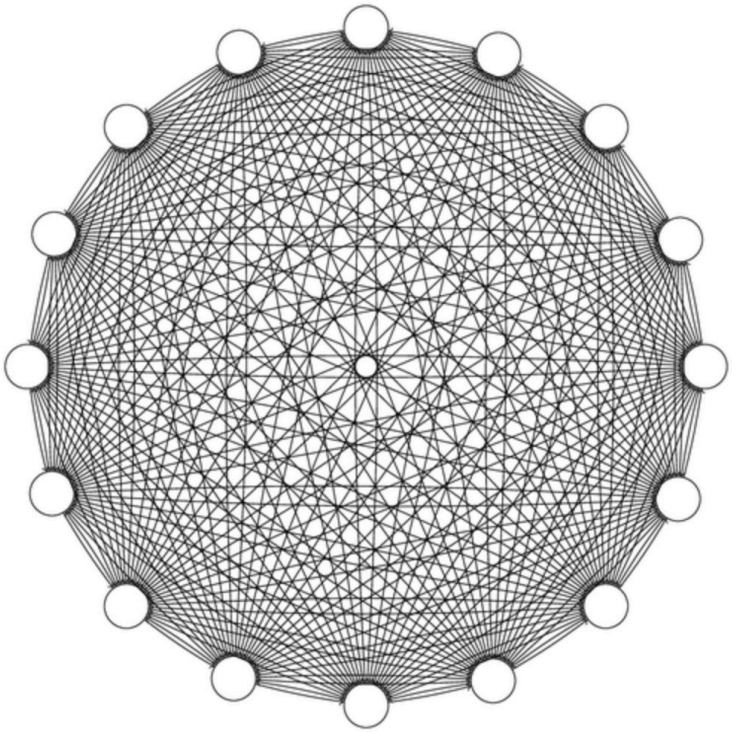
- Number of nodes:
  - "Tiny" map: 24
  - "Alley" map: 51

- Not fully connected, but number of nodes makes enumeration infeasible



What our TSP graph really looks like

- If we can't know distances between nodes without BFS/UCS/A*, then every spot that is not a wall is a node
- Number of nodes:
  - "Tiny" map: 24
  - "Alley" map: 51
- Not fully connected, but number of nodes makes enumeration infeasible



This is why we must do UCS or A*

(plus, you won't get credit for doing anything else)

Topic 2: How to control Pac-Man

Software Control Strategy

- The *action()* method is called before each step that Pac-Man takes, including the first
- Output must be a facing direction (NSEW) for the next move
- If you direct Pac-Man into a wall, the simulation ends

Pac-Man Control Agents

- Reflex agent
  - computes facing direction fresh every time
- Replanning agent
  - computes short-term plan then follows it
  - computes new short-term plan when first plan completed
- Planning agent
  - computes one (optimal) complete plan
  - follows the plan from start to finish

The Agent for Program 1

- Must be a planning agent
- Must compute entire plan before first move (using UCS)
- Must thereafter follow the plan, including for first move

The Agent for Program 1

- Must be a planning agent
- Must compute entire plan before first move (using A*)
- Must thereafter follow the plan, including for first move

Q: How does action() know when it's the first move?

The Agent for Program 1

- Must be a planning agent
- Must compute entire plan before first move (using A*)
- Must thereafter follow the plan, including for first move

Q: How does action() know when it's the first move?

A: Some data structure is not initialized (or could use boolean variable)

The Agent for Program 1

- Must be a planning agent
- Must compute entire plan before first move (using A*)
- Must thereafter follow the plan, including for first move

Q: How does action() know when it's the first move?

A: Some data structure is not initialized (or could use boolean variable)

Q: What does a plan look like?

Program 1 Agent

- Must be a planning agent
- Must compute entire plan before first move (using A*)
- Must thereafter follow the plan, including for first move

Q: How does action() know when it's the first move?

A: Some data structure is not initialized (or could use boolean variable)

Q: What does a plan look like?

A: An ordered sequence of adjacent cell locations or facing directions (locations probably easier)

Program 1 Agent

- Must be a planning agent
- Must compute entire plan before first move (using A*)
- Must thereafter follow the plan, including for first move

Q: How does action() know when it's the first move?

A: Some data structure is not initialized (or could use boolean variable)

Q: What does a plan look like?

A: An ordered sequence of adjacent cell locations or facing directions (locations probably easier)

Q: How do we follow such a plan?

Program 1 Agent

- Must be a planning agent
- Must compute entire plan before first move (using A*)
- Must thereafter follow the plan, including for first move

Q: How does action() know when it's the first move?

A: Some data structure is not initialized (or could use boolean variable)

Q: What does a plan look like?

A: An ordered sequence of adjacent cell locations or facing directions (locations probably easier)

Q: How do we follow such a plan?

A: Keep track of next-in-sequence, and compute facing direction from looking at current and next locations if sequence in form of locations

Search

Q: How much data should be propagated from each step in a partial plan to the next step?

Answer:

- Making many thousands of copies of the state array is too much baggage
- Think about how little data you really need

Q: In working up the solution with minimal data, how is one to know when all the food has been eaten?

Answer:

- From the initial state, find all the food and store their locations somewhere
- Then, as Pac-Man moves from position to position, count a food dot as eaten if Pac-Man lands in a food dot location.
- When all food dot locations have been touched by Pac-Man, search is done!

Questions?

pacsim

Interface PacAction

`public interface PacAction`

An interface for simulation experiments

Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
PacFace	action(java.lang.Object state) Choose the next direction in which Pac-Man should move
void	init() Initialization method to support restarting the simulation

Method Detail

action

`PacFace action(java.lang.Object state)`

Choose the next direction in which Pac-Man should move

Parameters:

`state - the cell array to examine`

Returns:

`a facing direction (or null)`

init

`void init()`

Initialization method to support restarting the simulation

pacsim

Class PacCell

java.lang.Object
    pacsim.PacCell

All Implemented Interfaces:

`java.io.Serializable, java.lang.Cloneable`

Direct Known Subclasses:

`FoodCell, GhostCell, GoalCell, HouseCell, PacmanCell, PathCell, PowerCell, StartCell, WallCell`

```
public class PacCell
extends java.lang.Object
implements java.lang.Cloneable, java.io.Serializable
```

Basic cell array object and superclass for other cell types

See Also:

`Serialized Form`

Field Summary

Fields

Modifier and Type	Field and Description
protected int	**cost**
protected java.awt.Point	**loc**

Constructor Summary

Constructors

Constructor and Description
PacCell(int x, int y)
PacCell(int x, int y, int cost)

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
`PacCell`	`clone()`
`int`	`getCost()` Return the cost value of this cell
`java.awt.Point`	`getLoc()` Return the location of this cell
`int`	`getX()` Return the x-coordinate this cell
`int`	`getY()` Return the y-coordinate this cell
`void`	`show(java.awt.Graphics g, int size)`

Methods inherited from class java.lang.Object

`equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

loc

```
protected java.awt.Point loc
```

cost

```
protected int cost
```

Constructor Detail

PacCell

```
public PacCell(int x,
               int y)
```

PacCell

```
public PacCell(int x,
               int y,
               int cost)
```

Method Detail

clone

```
public PacCell clone()
```

Overrides:

`clone in class java.lang.Object`

show

```
public void show(java.awt.Graphics g,
                 int size)
```

getLoc

```
public java.awt.Point getLoc()
```

Return the location of this cell

Returns:

`point location (zero-based)`

getX

```
public int getX()
```

Return the x-coordinate this cell

Returns:

`zero-based integer`

getY

```
public int getY()
```

Return the y-coordinate this cell

Returns:

```
zero-based integer
```

getCost

```
public int getCost()
```

Return the cost value of this cell

Returns:

```
an integer
```

pacsim

Enum PacFace

java.lang.Object
    java.lang.Enum<PacFace>
        pacsim.PacFace

All Implemented Interfaces:

`java.io.Serializable, java.lang.Comparable<PacFace>`

```
public enum PacFace
extends java.lang.Enum<PacFace>
```

Enumeration of facing directions

Enum Constant Summary

Enum Constants

Enum Constant and Description
E
N
S
W

Method Summary

All Methods | **Static Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`static PacFace`	**valueOf**`(java.lang.String name)` Returns the enum constant of this type with the specified name.
`static PacFace[]`	**values**`()` Returns an array containing the constants of this enum type, in the order they are declared.

Methods inherited from class java.lang.Enum

clone, compareTo, equals, finalize, getDeclaringClass, hashCode, name, ordinal, toString, valueOf

Methods inherited from class java.lang.Object

getClass, notify, notifyAll, wait, wait, wait

Enum Constant Detail

N

```
public static final PacFace N
```

S

```
public static final PacFace S
```

E

```
public static final PacFace E
```

W

```
public static final PacFace W
```

Method Detail

values

```
public static PacFace[] values()
```

Returns an array containing the constants of this enum type, in the order they are declared. This method may be used to iterate over the constants as follows:

```
for (PacFace c : PacFace.values())
```

```
    System.out.println(c);
```

Returns:

an array containing the constants of this enum type, in the order they are declared

valueOf

```
public static PacFace valueOf(java.lang.String name)
```

Returns the enum constant of this type with the specified name. The string must match *exactly* an identifier used to declare an enum constant in this type. (Extraneous whitespace characters are not permitted.)

Parameters:

name - the name of the enum constant to be returned.

Returns:

the enum constant with the specified name

Throws:

java.lang.IllegalArgumentException - if this enum type has no constant with the specified name

java.lang.NullPointerException - if the argument is null

pacsim

Class PacUtils

java.lang.Object
    pacsim.PacUtils

```
public class PacUtils
extends java.lang.Object
```

Multi-modal AI Simulator Utilities

Constructor Summary

Constructors

Constructor and Description
PacUtils()

Method Summary

All Methods | **Static Methods** | **Concrete Methods**

Modifier and Type	Method and Description
static **PacFace**	**avoidTarget**(java.awt.Point p, java.awt.Point t, **PacCell**[][] cell) Choose an available direction that maximizes the distance from a given target
static double	**euclideanDistance**(int x1, int y1, int x2, int y2) Compute the Euclidean distance between two points
static double	**euclideanDistance**(java.awt.Point p1, java.awt.Point p2) Compute the Euclidean distance between two points
static **PacFace**	**euclideanShortestToTarget**(java.awt.Point curr, **PacFace** face, java.awt.Point target, **PacCell**[][] cell) Chose the available direction that most closely approaches a

	target, using the Euclidean distance measure, but not the opposite of the current direction NOTE: This method returns null if the only option is to reverse.
static java.util.List<java.awt.Point>	**findGhosts**(**PacCell**[][] state) Find all the ghosts on the current board
static **PacmanCell**	**findPacman**(**PacCell**[][] state) Find Pac-Man if he is on the board (for simulation experiments)
static **StartCell**	**findStart**(**PacCell**[][] state) Find the start cell, if any (for search problems)
static boolean	**foodRemains**(**PacCell**[][] state) Determine whether any food remains on the board
static boolean	**goody**(int x, int y, **PacCell**[][] c) Determine whether the current cell contains either food or a power pellet
static int	**manhattanDistance**(int x1, int y1, int x2, int y2) Compute the Manhattan distance between two point locations
static int	**manhattanDistance**(java.awt.Point p1, java.awt.Point p2) Compute the Manhattan distance between two point locations
static **PacFace**	**manhattanShortestToTarget**(java.awt.Point curr, **PacFace** face, java.awt.Point target, **PacCell**[][] cell) Chose the available direction that most closely approaches a target, using the Manhattan distance measure
static **GhostCell**	**nearestGhost**(java.awt.Point p, **PacCell**[][] cell) Find the nearest ghost, if any
static java.awt.Point	**nearestGoody**(java.awt.Point p, **PacCell**[][] cell) Find the nearest food or power pellet cell
static java.awt.Point	**nearestGoodyButNot**(java.awt.Point p, java.awt.Point tgt, **PacCell**[][] cell) Find the nearest food or power pellet cell, but not a particular goody
static java.awt.Point	**nearestUnoccupied**(java.awt.Point p, **PacCell**[][] cell)

	Find the nearest unoccupied cell
static **PacCell**	**neighbor**(**PacFace** face, **PacCell** pc, **PacCell**[][] cell) Find the immediate neighbor of a given cell in a particular direction
static boolean	**oppositeFaces**(**PacFace** a, **PacFace** b) Determine whether two facing directions are opposites
static **PacFace**	**randomNotReverse**(java.awt.Point curr, **PacFace** face, java.awt.Point target, **PacCell**[][] cell) Choose a random available direction but not the opposite of the current direction
static **PacFace**	**randomOpenForGhost**(java.awt.Point curr, **PacCell**[][] cell) Choose a random direction where the next cell is not a ghost, wall, or Pac-Man
static **PacFace**	**randomOpenForPacman**(java.awt.Point curr, **PacCell**[][] cell) Choose a random facing direction that is not in the direction of a ghost, house, or wall cell
static **PacFace**	**reverse**(**PacFace** face) Find the opposite facing direction
static boolean	**unoccupied**(int x, int y, **PacCell**[][] c) Determine whether a particular cell is unoccupied

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

PacUtils

```
public PacUtils()
```

Method Detail

findStart

```
public static StartCell findStart(PacCell[][] state)
```

Find the start cell, if any (for search problems)

Parameters:

`state - the cell array to examine`

Returns:

`the Start Cell, if any`

findPacman

```
public static PacmanCell findPacman(PacCell[][] state)
```

Find Pac-Man if he is on the board (for simulation experiments)

Parameters:

`state - the cell array to examine`

Returns:

`the Pac-Man cell, if any`

findGhosts

```
public static java.util.List<java.awt.Point> findGhosts(PacCell[][] state)
```

Find all the ghosts on the current board

Parameters:

`state - the cell array to examine`

Returns:

`a list containing the ghost cells, if any`

foodRemains

```
public static boolean foodRemains(PacCell[][] state)
```

Determine whether any food remains on the board

Parameters:

`state - the cell array to examine`

Returns:

`T/F`

neighbor

```
public static PacCell neighbor(PacFace face,
                               PacCell pc,
                               PacCell[][] cell)
```

Find the immediate neighbor of a given cell in a particular direction

Parameters:

`face - the current direction`

`pc - the current cell`

`cell - the cell array to examine`

Returns:

`the immediate neighbor of the cell in the input direction, if any`

manhattanDistance

```
public static int manhattanDistance(java.awt.Point p1,
                                    java.awt.Point p2)
```

Compute the Manhattan distance between two point locations

Parameters:

`p1 - the first point`

`p2 - the second point`

Returns:

`non-negative integer distance`

manhattanDistance

```
public static int manhattanDistance(int x1,
                                    int y1,
                                    int x2,
                                    int y2)
```

Compute the Manhattan distance between two point locations

Parameters:

`x1 - x-coordinate of first point`

`y1 - y-coordinate of first point`

`x2 - x-coordinate of second point`

`y2 - y-coordinate of second point`

Returns:

non-negative integer distance

manhattanShortestToTarget

```
public static PacFace manhattanShortestToTarget(java.awt.Point curr,
                                                PacFace face,
                                                java.awt.Point target,
                                                PacCell[][] cell)
```

Chose the available direction that most closely approaches a target, using the Manhattan distance measure

Parameters:

curr - the current location

face - the current facing direction

target - the target location

cell - the cell array to examine

Returns:

a facing direction

euclideanDistance

```
public static double euclideanDistance(java.awt.Point p1,
                                       java.awt.Point p2)
```

Compute the Euclidean distance between two points

Parameters:

p1 - the first point

p2 - the second point

Returns:

a real-valued distance

euclideanDistance

```
public static double euclideanDistance(int x1,
                                       int y1,
                                       int x2,
                                       int y2)
```

Compute the Euclidean distance between two points

Parameters:

x1 - x-coordinate of first point

y1 - y-coordinate of first point

x2 - x-coordinate of second point

y2 - y-coordinate of second point

Returns:

a real-valued distance

euclideanShortestToTarget

```
public static PacFace euclideanShortestToTarget(java.awt.Point curr,
                                                PacFace face,
                                                java.awt.Point target,
                                                PacCell[][] cell)
```

Chose the available direction that most closely approaches a target, using the Euclidean distance measure, but not the opposite of the current direction NOTE: This method returns null if the only option is to reverse. In such case, it is usually best to reverse direction and then call this method again.

Parameters:

curr - the current location

face - the current facing direction

target - the target location

cell - the cell array to examine

Returns:

a facing direction

avoidTarget

```
public static PacFace avoidTarget(java.awt.Point p,
                                  java.awt.Point t,
                                  PacCell[][] cell)
```

Choose an available direction that maximizes the distance from a given target

Parameters:

p - the current location

t - the target location

cell - the cell array to examine

Returns:

a facing direction

randomNotReverse

```
public static PacFace randomNotReverse(java.awt.Point curr,
                                       PacFace face,
                                       java.awt.Point target,
                                       PacCell[][] cell)
```

Choose a random available direction but not the opposite of the current direction

Parameters:

`curr - the current cell location`

`face - the current facing direction`

`target - this parameter is not used`

`cell - the cell array to examine`

Returns:

`a facing direction`

randomOpenForPacman

```
public static PacFace randomOpenForPacman(java.awt.Point curr,
                                          PacCell[][] cell)
```

Choose a random facing direction that is not in the direction of a ghost, house, or wall cell

Parameters:

`curr - the current cell location`

`cell - the cell array to examine`

Returns:

`a facing direction`

randomOpenForGhost

```
public static PacFace randomOpenForGhost(java.awt.Point curr,
                                         PacCell[][] cell)
```

Choose a random direction where the next cell is not a ghost, wall, or Pac-Man

Parameters:

`curr - the current location`

`cell - the cell array to examine`

Returns:

`a facing direction`

nearestGoody

```
public static java.awt.Point nearestGoody(java.awt.Point p,
                                           PacCell[][] cell)
```

Find the nearest food or power pellet cell

Parameters:

`p - the current location`

`cell - the cell array to examine`

Returns:

`the location of the nearest goody`

nearestGoodyButNot

```
public static java.awt.Point nearestGoodyButNot(java.awt.Point p,
                                                 java.awt.Point tgt,
                                                 PacCell[][] cell)
```

Find the nearest food or power pellet cell, but not a particular goody

Parameters:

`p - the current location`

`tgt - the goody to avoid`

`cell - the cell array to examine`

Returns:

`the location of the nearest goody`

goody

```
public static boolean goody(int x,
                            int y,
                            PacCell[][] c)
```

Determine whether the current cell contains either food or a power pellet

Parameters:

`x - the x-coordinate of the current cell`

`y - the y-coordinate of the current cell`

`c - the cell array to examine`

Returns:

`T/F`

nearestGhost

```
public static GhostCell nearestGhost(java.awt.Point p,
                                     PacCell[][] cell)
```

Find the nearest ghost, if any

Parameters:

`p - the current location`

`cell - the cell array to examine`

Returns:

`the nearest ghost`

nearestUnoccupied

```
public static java.awt.Point nearestUnoccupied(java.awt.Point p,
                                               PacCell[][] cell)
```

Find the nearest unoccupied cell

Parameters:

`p - the current cell location`

`cell - the cell array to examine`

Returns:

`the nearest unoccupied cell`

unoccupied

```
public static boolean unoccupied(int x,
                                 int y,
                                 PacCell[][] c)
```

Determine whether a particular cell is unoccupied

Parameters:

`x - the x-coordinate of the input cell`

`y - the y-coordinate of the input cell`

`c - the input cell array`

Returns:

`T/F`

oppositeFaces

```
public static boolean oppositeFaces(PacFace a,
                                    PacFace b)
```

Determine whether two facing directions are opposites

Parameters:

`a - the first facing direction`

`b - the second facing direction`

Returns:

`T/F`

reverse

```
public static PacFace reverse(PacFace face)
```

Find the opposite facing direction

Parameters:

`face - the input facing direction`

Returns:

`the opposite direction of face`