

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
-----------------------------

<b>PacUtils</b> ()
--------------------

### Method Summary

All Methods

Static Methods

Concrete Methods

Modifier and Type	Method and Description
static <b>PacFace</b>	<b>anyRandomForGhost</b> (java.awt.Point curr, <b>PacCell</b> [] [] cell) Choose a random direction where the next cell is not a ghost or wall cell NOTE: this method should be used when in CHASE or SCATTER mode,
static void	<b>appendPointList</b> (java.util.List<java.awt.Point> a, java.util.List<java.awt.Point> b) Append one point list to another.
static <b>PacFace</b>	<b>avoidTarget</b> (java.awt.Point p, java.awt.Point t, <b>PacCell</b> [][] cell) Choose an available direction that maximizes the distance from a given target
static <b>PacCell</b> [][]	<b>cloneGrid</b> ( <b>PacCell</b> [][] array) Clone a PacCell grid
static java.util.List<java.awt.Point>	<b>clonePointList</b> (java.util.List<java.awt.Point> list) Clone a list of Point objects
static <b>PacFace</b>	<b>direction</b> (java.awt.Point p, java.awt.Point q) Determine the facing direction from one cell to another

static <b>PacCell</b>	<b>distantNeighbor</b> ( <b>PacFace</b> face, int d, <b>PacCell</b> pc, <b>PacCell</b> [][] cell) Find the neighbor cell at a given distance in a particular direction
static double	<b>euclideanDistance</b> (int x1, int y1, int x2, int y2) Compute the Euclidean distance between two points
static double	<b>euclideanDistance</b> (java.awt.Point p1, java.awt.Point p2) Compute the Euclidean distance between two points
static <b>PacFace</b>	<b>euclideanShortestToTarget</b> (java.awt.Point curr, <b>PacFace</b> face, java.awt.Point target, <b>PacCell</b> [][] cell) Chose the available direction that most closely approaches a target, using the Euclidean distance measure, but not the opposite of the current direction; ties are broken randomly NOTE: This method returns null if the only option is to reverse.
static java.util.List<java.awt.Point>	<b>findFood</b> ( <b>PacCell</b> [][] state) Find the locations of all remaining food, including what may be covered up by a ghost
static java.util.List<java.awt.Point>	<b>findFood</b> ( <b>PacCell</b> [][] state, <b>PacTeam</b> team) Find the locations of all the remaining food for a team.
static java.util.List<java.awt.Point>	<b>findGhosts</b> ( <b>PacCell</b> [][] state) Find all the ghosts on the current board
static java.util.List<java.awt.Point>	<b>findMorphs</b> ( <b>PacCell</b> [][] state) Find the locations of all the morphs on the current board
static java.util.List<java.awt.Point>	<b>findMorphs</b> ( <b>PacCell</b> [][] state, <b>PacTeam</b> team) Find the locations of all morphs for a particular team
static <b>PacmanCell</b>	<b>findPacman</b> ( <b>PacCell</b> [][] state) Find Pac-Man if he is on the board (for simulation experiments)
static <b>StartCell</b>	<b>findStart</b> ( <b>PacCell</b> [][] state) Find the start cell, if any (for search problems)
static boolean	<b>food</b> (int x, int y, <b>PacCell</b> [][] c) Determine whether the current cell contains a food pellet
static boolean	<b>foodRemains</b> ( <b>PacCell</b> [][] state) Determine whether any food remains on the board
static boolean	<b>foodRemains</b> ( <b>PacCell</b> [][] state, <b>PacTeam</b> team) Determine whether any food belonging to a given team remains on the board
static java.util.List< <b>MorphCell</b> >	<b>getMorphs</b> ( <b>PacCell</b> [][] state) Retrieve all the morphs on the current board, if any
static java.util.List< <b>MorphCell</b> >	<b>getMorphs</b> ( <b>PacCell</b> [][] state, <b>PacTeam</b> team) Retrieve all morphs, if any, for a given team
static boolean	<b>goody</b> (int x, int y, <b>PacCell</b> [][] c)

	Determine whether the current cell contains either food or a power pellet
static int	<b>manhattanDistance</b> (int x1, int y1, int x2, int y2) Compute the Manhattan distance between two point locations
static int	<b>manhattanDistance</b> (java.awt.Point p1, java.awt.Point p2) Compute the Manhattan distance between two point locations
static <b>PacFace</b>	<b>manhattanShortestToTarget</b> (java.awt.Point curr, <b>PacFace</b> face, java.awt.Point target, <b>PacCell</b> [][] cell) Chose the available direction that most closely approaches a target, using the Manhattan distance measure, but not the opposite of the current direction; ties are broken randomly
static java.util.List<java.awt.Point>	<b>morphHomes</b> ( <b>PacCell</b> [][] state, <b>PacTeam</b> team) Find the locations of all morph homes for a particular team
static <b>PacCell</b> [][]	<b>moveGhost</b> (java.awt.Point curr, java.awt.Point next, <b>PacCell</b> [][] array) Move a ghost on an input grid This method does nothing if a ghost cannot be found at location curr or if next is not immediately adjacent.
static <b>PacCell</b> [][]	<b>movePacman</b> (java.awt.Point curr, java.awt.Point next, <b>PacCell</b> [][] array) Move Pacman on an input grid This method does nothing if Pacman cannot be found at location curr or if next is not immediately adjacent.
static java.awt.Point	<b>nearestFood</b> ( <b>PacCell</b> [][] grid, java.awt.Point mloc, <b>PacTeam</b> team) Find the nearest food cell belonging to a given team
static java.awt.Point	<b>nearestFood</b> (java.awt.Point p, <b>PacCell</b> [][] cell) Find the nearest food pellet, if any, using the city block measure
static <b>GhostCell</b>	<b>nearestGhost</b> (java.awt.Point p, <b>PacCell</b> [][] cell) Find the nearest ghost, if any
static java.awt.Point	<b>nearestGoody</b> (java.awt.Point p, <b>PacCell</b> [][] cell) Find the nearest food or power pellet cell, if any
static java.awt.Point	<b>nearestGoodyButNot</b> (java.awt.Point p, java.awt.Point tgt, <b>PacCell</b> [][] cell) Find the nearest food or power pellet cell, but not a particular goody
static java.awt.Point	<b>nearestMorph</b> ( <b>PacCell</b> [][] grid, java.awt.Point mloc, <b>PacTeam</b> team) Find the nearest opponent
static java.awt.Point	<b>nearestPower</b> (java.awt.Point p, <b>PacCell</b> [][] cell) Find the nearest power cell, if any
static java.awt.Point	<b>nearestUnoccupied</b> (java.awt.Point p, <b>PacCell</b> [][] cell)

	Find the nearest unoccupied cell; if cannot find one, then choose a random unoccupied cell
static <b>PacCell</b>	<b>neighbor</b> ( <b>PacFace</b> face, <b>PacCell</b> pc, <b>PacCell</b> [][] cell) Find the immediate neighbor of a given cell in a particular direction
static <b>PacCell</b>	<b>neighbor</b> ( <b>PacFace</b> face, java.awt.Point p, <b>PacCell</b> [][] cell) Find the immediate neighbor of a given cell location in a particular direction
static int	<b>numFood</b> ( <b>PacCell</b> [][] state) Determine how many food dots remain on the board
static int	<b>numFood</b> ( <b>PacCell</b> [][] state, <b>PacTeam</b> team) Determine how many food dots belonging to a given team remain on the board
static int	<b>numPower</b> ( <b>PacCell</b> [][] state) Determine how many power pellets remain on the board
static <b>PacTeam</b>	<b>opposingTeam</b> ( <b>PacTeam</b> team) Determine the opposing team color
static <b>PacFace</b>	<b>oppositeFace</b> ( <b>PacFace</b> face) Find the opposite facing direction
static boolean	<b>oppositeFaces</b> ( <b>PacFace</b> a, <b>PacFace</b> b) Determine whether two facing directions are opposites
static boolean	<b>power</b> (int x, int y, <b>PacCell</b> [][] c) Determine whether the current cell contains a power pellet
static <b>PacFace</b>	<b>randomFace</b> () Get a single random facing direction
static <b>PacFace</b> []	<b>randomFaces</b> () Get all facing directions in random order
static <b>PacFace</b>	<b>randomNotReverse</b> (java.awt.Point curr, <b>PacFace</b> face, java.awt.Point target, <b>PacCell</b> [][] cell) Choose a random available direction but not the opposite of the current direction
static <b>PacFace</b>	<b>randomOpenForGhost</b> (java.awt.Point curr, <b>PacCell</b> [][] cell) Choose a random direction where the next cell is not a ghost, wall, or Pac-Man NOTE: this method should be used when in FEAR mode (so can't go to Pac-Man cell)
static <b>PacFace</b>	<b>randomOpenForPacman</b> (java.awt.Point curr, <b>PacCell</b> [][] cell) Choose a random facing direction that is not in the direction of a ghost, house, or wall cell
static <b>PacFace</b>	<b>reverse</b> ( <b>PacFace</b> face) Find the opposite facing direction
static boolean	<b>unoccupied</b> (int x, int y, <b>PacCell</b> [][] 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

appendPointList

```
public static void appendPointList(java.util.List<java.awt.Point> a,
                                   java.util.List<java.awt.Point> b)
```

Append one point list to another. This method consumes the second input list.

Parameters:

- a - - the point list that will have the other list appended to it
- b - - the point list that will be appended to the first point list

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

**findMorphs**

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

Find the locations of all the morphs on the current board

**Parameters:**

state – the cell array to examine

**Returns:**

a list containing the morph cells, if any

**getMorphs**

```
public static java.util.List<MorphCell> getMorphs(PacCell[][] state)
```

Retrieve all the morphs on the current board, if any

**Parameters:**

state – the cell array to examine

**Returns:**

a list containing the morph cells, if any

**findMorphs**

```
public static java.util.List<java.awt.Point> findMorphs(PacCell[][] state,
                                                         PacTeam team)
```

Find the locations of all morphs for a particular team

**Parameters:**

state – the cell array to examine

team – the team whose morphs to examine

**Returns:**

a list containing the morph cells, if any

**morphHomes**

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

PacTeam team)

Find the locations of all morph homes for a particular team

**Parameters:**

state – the cell array to examine

team – the team whose morph homes to find

**Returns:**

a list containing the morph cells, if any

**getMorphs**

```
public static java.util.List<MorphCell> getMorphs(PacCell[][] state,
                                                    PacTeam team)
```

Retrieve all morphs, if any, for a given team

**Parameters:**

state – the cell array to examine

team – the team whose morphs to retrieve

**Returns:**

a list containing the morph 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

**foodRemains**

```
public static boolean foodRemains(PacCell[][] state,
                                    PacTeam team)
```

Determine whether any food belonging to a given team remains on the board

**Parameters:**

state – the cell array to examine

team – the team whose food is being examined

**Returns:**

T/F

**findFood**



```
public static java.util.List<java.awt.Point> findFood(PacCell[][] state,
                                                    PacTeam team)
```

Find the locations of all the remaining food for a team.

**Parameters:**

state – the cell array to examine

team – the team whose food is being examined

**Returns:**

list of locations of remaining food for team

**findFood**

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

Find the locations of all remaining food, including what may be covered up by a ghost

**Parameters:**

state – the cell array to examine

**Returns:**

list of locations of remaining food

**numFood**

```
public static int numFood(PacCell[][] state)
```

Determine how many food dots remain on the board

**Parameters:**

state – the cell array to examine

**Returns:**

number of remaining food dots

**numFood**

```
public static int numFood(PacCell[][] state,
                        PacTeam team)
```

Determine how many food dots belonging to a given team remain on the board

**Parameters:**

state – the cell array to examine

team – the team whose food dots to count

**Returns:**

number of remaining food dots

**numPower**

```
public static int numPower(PacCell[][] state)
```



Determine how many power pellets remain on the board

**Parameters:**

state - the cell array to examine

**Returns:**

number of remaining power pellets

**distantNeighbor**

```
public static PacCell distantNeighbor(PacFace face,
                                     int d,
                                     PacCell pc,
                                     PacCell[][] cell)
```

Find the neighbor cell at a given distance in a particular direction

**Parameters:**

face - the current direction

d - the number of cells distant

pc - the current cell

cell - the cell array to examine

**Returns:**

the cell at the given distance and direction; this method returns null if the location is off the board

**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

**neighbor**

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

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

**Parameters:**

face - the current direction

p - the current cell location

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, but not the opposite of the current direction; ties are broken randomly

**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; ties are broken randomly 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 NOTE: this method should be used when in FEAR mode (so can't go to Pac-Man cell)

Parameters:

curr - the current location

cell - the cell array to examine

Returns:

a facing direction

anyRandomForGhost

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

Choose a random direction where the next cell is not a ghost or wall cell NOTE: this method should be used when in CHASE or SCATTER mode,

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, if any

Parameters:

p - the current location

cell - the cell array to examine

Returns:

the location of the nearest goody, or null

nearestFood

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

Find the nearest food pellet, if any, using the city block measure

**Parameters:**

p - the current location

cell - the cell array to examine

**Returns:**

the location of the nearest food, or null

**nearestPower**

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

Find the nearest power cell, if any

**Parameters:**

p - the current location

cell - the cell array to examine

**Returns:**

the location of the nearest power cell, or null

**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

**food**

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

Determine whether the current cell contains a food 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

**power**

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

Determine whether the current cell contains 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; if cannot find one, then choose a random unoccupied cell

Parameters:

p - the current cell location

cell - the cell array to examine

Returns:

the nearest or random 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



Move a ghost on an input grid This method does nothing if a ghost cannot be found at location curr or if next is not immediately adjacent. Next must not be a wall cell or another ghost. This method preserves the underlying base costs and types for all cells moved into and restores the underlying base cell for curr.

**Parameters:**

curr - current ghost position

next - next ghost position

array - the input grid

**Returns:**

grid, the resulting grid after the move

**randomFace**

```
public static PacFace randomFace()
```

Get a single random facing direction

**Returns:**

a random facing direction

**randomFaces**

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

Get all facing directions in random order

**Returns:**

array of facing directions in random order

**direction**

```
public static PacFace direction(java.awt.Point p,
                                java.awt.Point q)
```

Determine the facing direction from one cell to another

**Parameters:**

p - starting cell location

q - ending cell location

**Returns:**

facing direction from starting cell to ending cell

**opposingTeam**

```
public static PacTeam opposingTeam(PacTeam team)
```

Determine the opposing team color

**Parameters:**

team - this team's color

**Returns:**  
opponents team color

**nearestFood**

```
public static java.awt.Point nearestFood(PacCell[][] grid,
                                         java.awt.Point mloc,
                                         PacTeam team)
```

Find the nearest food cell belonging to a given team

**Parameters:**

- grid - the cell array to examine
- mloc - the current location
- team - the team whose food to examine

**Returns:**

the cell location of the nearest food to the current location

**nearestMorph**

```
public static java.awt.Point nearestMorph(PacCell[][] grid,
                                         java.awt.Point mloc,
                                         PacTeam team)
```

Find the nearest opponent

**Parameters:**

- grid - the cell array to examine
- mloc - the current location
- team - the team whose morphs to examine

**Returns:**

the cell location of the nearest morph to the current location

**oppositeFace**

```
public static PacFace oppositeFace(PacFace face)
```

Find the opposite facing direction

**Parameters:**

- face - the current facing direction

**Returns:**

the opposite facing direction

