PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

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` | `anyRandomForGhost(java.awt.Point curr, PacCell[][] cell)`<br>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 PacFace` | `avoidTarget(java.awt.Point p, java.awt.Point t, PacCell[][] cell)`<br>Choose an available direction that maximizes the distance from a given target |
| `static PacCell[][]` | `cloneGrid(PacCell[][] array)`<br>Clone a PacCell grid |
| `static java.util.List<java.awt.Point>` | `clonePointList(java.util.List<java.awt.Point> list)`<br>Clone a list of Point objects |
| `static double` | `euclideanDistance(int x1, int y1, int x2, int y2)`<br>Compute the Euclidean distance between two points |
| `static double` | `euclideanDistance(java.awt.Point p1, java.awt.Point p2)`<br>Compute the Euclidean distance between two points |

| | |
|---|---|
| static PacFace | euclideanShortestToTarget(java.awt.Point curr, PacFace face, java.awt.Point target, PacCell[][] cell)<br>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)<br>Find all the ghosts on the current board |
| static PacmanCell | findPacman(PacCell[][] state)<br>Find Pac-Man if he is on the board (for simulation experiments) |
| static StartCell | findStart(PacCell[][] state)<br>Find the start cell, if any (for search problems) |
| static boolean | food(int x, int y, PacCell[][] c)<br>Determine whether the current cell contains a food pellet |
| static boolean | foodRemains(PacCell[][] state)<br>Determine whether any food remains on the board |
| static boolean | goody(int x, int y, PacCell[][] c)<br>Determine whether the current cell contains either food or a power pellet |
| static int | manhattanDistance(int x1, int y1, int x2, int y2)<br>Compute the Manhattan distance between two point locations |
| static int | manhattanDistance(java.awt.Point p1, java.awt.Point p2)<br>Compute the Manhattan distance between two point locations |
| static PacFace | manhattanShortestToTarget(java.awt.Point curr, PacFace face, java.awt.Point target, PacCell[][] cell)<br>Chose the available direction that most closely approaches a target, using the Manhattan distance measure |
| static PacCell[][] | moveGhost(java.awt.Point curr, java.awt.Point next, PacCell[][] array)<br>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 PacCell[][] | movePacman(java.awt.Point curr, java.awt.Point next, PacCell[][] array)<br>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 | nearestFood(java.awt.Point p, PacCell[][] cell)<br>Find the nearest food pellet, if any |
| static GhostCell | nearestGhost(java.awt.Point p, PacCell[][] cell)<br>Find the nearest ghost, if any |

| | |
|---|---|
| `static java.awt.Point` | **`nearestGoody`**`(java.awt.Point p, `**`PacCell`**`[][] cell)`<br>Find the nearest food or power pellet cell, if any |
| `static java.awt.Point` | **`nearestGoodyButNot`**`(java.awt.Point p, java.awt.Point tgt, `**`PacCell`**`[][] cell)`<br>Find the nearest food or power pellet cell, but not a particular goody |
| `static java.awt.Point` | **`nearestPower`**`(java.awt.Point p, `**`PacCell`**`[][] cell)`<br>Find the nearest power cell, if any |
| `static java.awt.Point` | **`nearestUnoccupied`**`(java.awt.Point p, `**`PacCell`**`[][] cell)`<br>Find the nearest unoccupied cell; if cannot find one, then choose a random unoccupied cell |
| `static `**`PacCell`** | **`neighbor`**`(`**`PacFace`**` face, `**`PacCell`**` pc, `**`PacCell`**`[][] cell)`<br>Find the immediate neighbor of a given cell in a particular direction |
| `static `**`PacCell`** | **`neighbor`**`(`**`PacFace`**` face, java.awt.Point p, `**`PacCell`**`[][] cell)`<br>Find the immediate neighbor of a given cell location in a particular direction |
| `static int` | **`numFood`**`(`**`PacCell`**`[][] state)`<br>Determine how many food dots remain on the board |
| `static int` | **`numPower`**`(`**`PacCell`**`[][] state)`<br>Determine how many power pellets remain on the board |
| `static boolean` | **`oppositeFaces`**`(`**`PacFace`**` a, `**`PacFace`**` b)`<br>Determine whether two facing directions are opposites |
| `static boolean` | **`power`**`(int x, int y, `**`PacCell`**`[][] c)`<br>Determine whether the current cell contains a power pellet |
| `static `**`PacFace`** | **`randomNotReverse`**`(java.awt.Point curr, `**`PacFace`**` face, java.awt.Point target, `**`PacCell`**`[][] cell)`<br>Choose a random available direction but not the opposite of the current direction |
| `static `**`PacFace`** | **`randomOpenForGhost`**`(java.awt.Point curr, `**`PacCell`**`[][] cell)`<br>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 `**`PacFace`** | **`randomOpenForPacman`**`(java.awt.Point curr, `**`PacCell`**`[][] cell)`<br>Choose a random facing direction that is not in the direction of a ghost, house, or wall cell |
| `static `**`PacFace`** | **`reverse`**`(`**`PacFace`**` face)`<br>Find the opposite facing direction |
| `static boolean` | **`unoccupied`**`(int x, int y, `**`PacCell`**`[][] c)`<br>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

### 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

### 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

### 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

**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 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

**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

**Returns:**

the opposite direction of face

### cloneGrid

```
public static PacCell[][] cloneGrid(PacCell[][] array)
```

Clone a PacCell grid

**Parameters:**

array - the input grid

**Returns:**

a clone of the input

### clonePointList

```
public static java.util.List<java.awt.Point> clonePointList(java.util.List<java.awt.Point> list)
```

Clone a list of Point objects

**Parameters:**

list - input list of Points

**Returns:**

newList, the cloned list

### movePacman

```
public static PacCell[][] movePacman(java.awt.Point curr,
                                     java.awt.Point next,
                                     PacCell[][] 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. Next must not be a wall cell. If next is occupied by a fearful ghost, this method moves it to its home cell, or if occupied, to the nearest unoccupied cell. If the next cell is a power pellet, this method sets all ghosts to fearful, effectively resetting the fear timer if they are already afraid. If the next cell is occupied by a non-fearful ghost, no move is made. This method preserves the underlying base costs and types for all cells moved into.

**Parameters:**

curr - current Pacman position

next - next Pacman position

array - the input grid

**Returns:**

grid, the resulting grid after the move

### moveGhost

```
public static PacCell[][] moveGhost(java.awt.Point curr,
                                    java.awt.Point next,
                                    PacCell[][] 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. 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`