CLASS DESIGN FOR ALLOCATION LIBRARY

# OVERALL NOTES

## TERMINOLOGY

* An assignment is a specified amount of time for which a given resource searches a given area.
* An allocation is a set of assignments.

## ERRORS

Where necessary, throw exceptions to indicate errors. (For example, throw an exception if one of getAssignmentsForResource() / getAssignmentsForArea() / getAssignedAreas() / getCoverage() / getPOD() / getPOS() / getNewPOC() / getTotalPOS() are called, but no allocation has been set or calculated.)

## TESTING

Have a compiler tag named \_classname*\_*testmode for each class.

If this tag is included during compilation, additional output statements (of debugging information) will be made within the functions of the class. Debugging output will include a statement upon entry to every function that indicates which function has been entered (and displays the values of the parameters if they are of a type that can easily be displayed).

Some testing functions will also be included during compilation only if this tag is defined. For example, the Washburn class will need a member testing function which is able to directly set a configuration of the Washburn tree (by altering the (private) data members) and then call a private member function to check that it operates correctly upon the tree. The displayTree() member function (also included only if the tag is defined) would be used to display the tree both before and after each private function was called, proving that each operated correctly.

## UNITS FOR DATA

No units are specified for any quantities, as the Allocation class does not enforce any particular units. The only requirements is that the units used be consistent.

## EXPECTED SIZE OF INPUT DATA

The class is designed with maximums of approximately 20 resources and 1000 areas in mind. However, there is no upper limit enforced by the class itself. (Although available memory may, of course, impose its own constraint.)

## RESOURCE AND AREA NUMBERS

Resource and area numbers are assumed to be in the ranges 0,…,numResources-1 and 0,…,numAreas –1 (rather than 1,…,numResources and 1,…,numAreas).

# CLASS: Allocation

## NOTES

This class is used to perform calculations related to allocations (of resources to areas for searching) only. For example, it can be used to calculate coverage and POD values for a given allocation, but cannot be used to calculate a POD from an arbitrary coverage. At present, it deals with random searches (exponential detection function) only.

Note that this class may be used to both calculate an optimal allocation and assess the success of an arbitrarily specified allocation:

* To calculate an optimal allocation: specify (estimated) effectiveness and POC values in constructor and call calcAllocation() with the search endurance (time available) for each resource. getAssignmentsForResource(), getAssignmentsForArea() and getAssignedAreas() may then be used to obtain the assignments, whilst getCoverage(), getPOD(), getPOS(), getNewPOC() and getTotalPOS() may be used to determine the success of the allocation.
* To evaluate the success of an arbitrary allocation: specify (known) effectiveness (from measured speed and sweep width) and POC values in constructor and call setAllocation() with the actual amount of time for which each resource searched each area. getCoverage(), getPOD(), getPOS(), getNewPOC() and getTotalPOS() may be used to determine the success of the allocation.

## PLUG-IN ALLOCATION ALGORITHMS

The Allocation virtual class specifies the common interface that any actual allocator must have. Each possible allocation algorithm, such as Charnes-Cooper or Washburn, is implemented as a subclass of Allocation, which inherits and usually overrides the virtual methods.

## PUBLIC FUNCTIONS OF THE ALLOCATION CLASS

|  |  |
| --- | --- |
| SIGNATURE | DESCRIPTION |
| Allocation(int numResources, int numAreas, const double\* effectiveness, const double\* POC) | Generic/Default Constructor.**[ZZZ]**  [Not clear this is desired. But if it is generic, it would decide which algorithm to use and return one of those objects.  Deciding which algorithm to use:   * If WASHBURN is specified, or DEFAULT is specified and myNumResources ≥ 2, the Washburn algorithm is used. * If CC or DEFAULT is specified, and myNumResources = 1, the Charnes‑Cooper algorithm is used. * If CC is specified and myNumResources ≥ 2, an exception is thrown.   ]  Arguments are:   * Number of resources (≥1) * Number of areas (≥1) * Pointer to first element of 2D array of effectiveness values (double[numResources][numAreas])   Each value is ≥ 0.  The Allocation object stores a local copy of these values in myEffectiveness. **[ZZZ]**   * Pointer to first element of array of (initial) POC values (double[numAreas])   Each value is ≥ 0.  The Allocation object takes a local copy of these values in myPOC.  The constructor must check that all values are within the required ranges (as indicated above). |
| *algorithm*(int p\_no\_resources, int p\_no\_areas, const double \*p\_effectiveness, const double \*p\_POC) | Algorithm-specific constructor. For example, “CharnesCooper” or “Washburn”.  Arguments as above, but:   * some values may have default values, or even forced values. For example, CC can use only 1 resource. |
| virtual void calcAllocation(const double\* searchEndurance) | Arguments is a pointer to first element of array of search endurance values (double[myNumResources]). Each value is ≥ 0 (this should be checked).  Each algorithm implements this in its own specific way. |
| virtual void setAllocation(const double\* assignment) | Directly sets the amount of time for which each resource searches each area. Argument is a pointer to the first element of the 2D array of assignment times (double[myNumResources][myNumAreas]).  Each algorithm implements this in its own specific way, allocating memory as needed to put the allocation into its data structure. |
| virtual AreaIterator getAssignmentsForResource(int resourceNum) | Returns an iterator which provides access to the areas that the specified resource is assigned to search. This iterator must provide the following functions:   * goToStart() (go to first item in list) * goToEnd() (go to one item past last in list) * atStart() (check if at start of list) * atEnd() (check if at end of list) * ++ (increment) * -- (decrement) * () (Return object currently pointed to by iterator. Return value is undefined if iterator has been moved beyond either end of the list.)   The underlying objects (which the iterator points to) are of type AreaAssignment and store the area number and time for an assignment. These objects provide the following functions:   * AreaAssignment(areaNum, time) (Area number and time specified at construction and cannot subsequently be changed.) * getAreaNum() * getTime()   Each algorithm implements this in its own way, over its own data structure. |
| virtual ResourceIterator getAssignmentsForArea(int areaNum) | Returns an iterator which provides access to the resources that are assigned to search the specified area. This iterator must provide the following functions:   * goToStart() (go to first item in list) * goToEnd() (go to one item past last in list) * atStart() (check if at start of list) * atEnd() (check if at end of list) * ++ (increment) * -- (decrement) * () (Return object currently pointed to by iterator. Return value is undefined if iterator has been moved beyond either end of the list.)   The underlying objects (which the iterator points to) are of type ResourceAssignment and store the resource number and time for an assignment. These objects provide the following functions:   * ResourceAssignment(resourceNum, time) (Resource number and time specified at construction and cannot subsequently be changed.) * getResourceNum() * getTime()   Each algorithm implments this in its own way, over its own data structure. |
| virtual IntIterator getAssignedAreas(void) | Returns an iterator which provides access to the areas that have at least one resource assigned to search them. This iterator must provide the following functions:   * goToStart() (go to first item in list) * goToEnd() (go to one item past last in list) * atStart() (check if at start of list) * atEnd() (check if at end of list) * ++ (increment) * -- (decrement) * () (Return object currently pointed to by iterator. Return value is undefined if iterator has been moved beyond either end of the list.)   The underlying objects (which the iterator points to) are integers. These are the area numbers of areas that have resources assigned to them.  Each algorithm implements this in its own way, over its own data structure. |
| double getCoverage(int areaNum) | Returns the coverage for the specified area for the set of assignments.   * If myCoverage == NULL, memory is allocated for myCoverage (an array double [myNumAreas]), then all coverage values are calculated and stored in this array (myCoverage[j] = calcCoverage(j)). * myCoverage[areaNum] is returned. |
| double getPOD(int areaNum) | Returns the POD for the specified area for the set of assignments.   * If myPOD == NULL, memory is allocated for myPOD (an array double [myNumAreas]), then all POD values are calculated and stored in this array (myPOD[j] = calcPOD(j)). * myPOD[areaNum] is returned. |
| double getPOS(int areaNum) | Returns the POS for the specified area for the set of assignments.   * If myPOS == NULL, memory is allocated for myPOS (an array double [myNumAreas]), then all POS values are calculated and stored in this array (myPOS[j] = calcPOS(j)). * myPOS[areaNum] is returned. |
| double getNewPOC(int areaNum) | Returns the new POC for the specified area for the set of assignments.   * If myNewPOC == NULL, memory is allocated for myNewPOC (an array double [myNumAreas]), then all new POC values are calculated and stored in this array (myNewPOC[j] = calcNewPOC(j)). * myNewPOC[areaNum] is returned. |
| double getTotalPOS(void) | Returns the total POS (for all areas) for the set of assignments.   * If myTotalPOS == NULL, memory is allocated for myTotalPOS (a single double) and the total POS is calculated:     and stored in \*myTotalPOS.  (POSj is the POS in area j)   * The value \*myTotalPOS is returned. |

## PRIVATE FUNCTIONS

|  |  |
| --- | --- |
| SIGNATURE | DESCRIPTION |
| double calcCoverage(int areaNum) | Returns the coverage obtained in the given area with the set of assignments.    (wij and Timeij are the effectiveness and time assigned for resource i in area j) |
| inline double calcPOD(int areaNum) | Returns the POD in the given area with the set of assignments (for a random search) (POD = 1 – exp(-C)) |
| inline double calcPOS(int areaNum) | Returns the POS in the given area with the set of assignments (POS = POD \* POC) |
| inline double calcNewPOC(int areaNum) | Returns the new POC in the given area with the set of assignments (newPOC = (1-POD) \* POC) |
| inline double calcInitialPSR(int resourceNum, int areaNum) | Returns the initial PSR for the given resource in the given area:  PSRij = wij \* POCj  (wij is the effectiveness of resource i in area j and POCj is the POC in area j) |
| double getEffectiveness(int resourceNum, int areaNum) | Returns the effectiveness of the specified resource in the specified area (myEffectiveness[resourceNum\*myNumAreas+areaNum]).  **[Shd this be public??]** |

## DATA MEMBERS [ALL PRIVATE]

|  |  |
| --- | --- |
| NAME | DESCRIPTION |
| myNumResources | const int: Number of resources for this allocation. |
| myNumAreas | const int: Number of areas for this allocation. |
| myPOC | double\* (dynamically allocate an array [myNumAreas] upon construction): Initial POC value for each area. |
| myEffectiveness | double\* (dynamically allocate an array [myNumResources\*myNumAreas] upon construction): Effectiveness values for each resource in each area. This 1D array represents a 2D table [myNumResources][myNumAreas], with myEffectiveness[r\*myNumAreas+a] being the effectiveness of resource r in area a. |
| myCoverage | double\* (dynamically allocate an array [myNumAreas] when needed): Coverage in each area for the set of assignments. |
| myPOD | double\* (dynamically allocate an array [myNumAreas] when needed): POD in each area for the set of assignments. |
| myPOS | double\* (dynamically allocate an array [myNumAreas] when needed): POS in each area for the set of assignments. |
| myNewPOC | double\* (dynamically allocate an array [myNumAreas] when needed): New POC in each area for the set of assignments. |
| myTotalPOS | double\*: Pointer to a (single) double which holds the total POS for the set of assignments. |

# CLASS: Charnes-Cooper

Probably best to read the code right now. And see the uml diagram. CC must implement the public functions. Here’s some notes from before:

Calculating an allocation using a given algorithm:

* Charnes‑Cooper algorithm:

Allocate memory for the array double myAssignments[myNumResources\*myNumAreas], then call calcAllocationCharnesCooper() to calculate the allocation and store it in myAssignments.

|  |  |
| --- | --- |
| void calcAllocationCharnesCooper(double searchEndurance) | Arguments is the time available for searching for the single resource (with resource number 0).  Uses the Charnes-Cooper algorithm to calculate the optimal allocation of the single resource to areas and stores the result in myAssignments.  If the resource is useless (in every area, either POC is zero or its effectiveness is zero), no allocation is made. |

# CLASS: Washburn

## NOTES

### Not fully updated to reflect the new plug-in structure. [--crt 15 Mar. 02]

Washburn will now look like Charnes-Cooper --- a subclass of allocation. It will just have a more complicated internal structure.

### Use of Washburn class

### Ensuring that no assignments are made for useless resources

If a resource is useless (in every area, either the POC is zero or the effectiveness of the resource is zero), then no assignments should be made for that resource. Fortunately, the Washburn algorithm should ensure that this occurs, so there is no need to add additional code to deal with this case:

* The artificial area created during the Washburn allocation process has a positive μ‑value (), which is equivalent to a positive POC.
* The effectiveness of all resources in the artificial area is positive:

 

 (where wij is the effectiveness of resource i in area j and Ti is the search endurance of resource i)

* The above points mean that all resources are able to effectively search the artificial area, although most resources will not do so because they will have higher POC and effectiveness values in other areas. However, useless resources are *only* able to effectively search the artificial area, so they should be assigned to search it during the Washburn allocation process. These assignments will then be ignored when the assignments are extracted from the Washburn tree using getSolution() – leaving the useless resources unassigned (as desired).

### Storing POC, effectiveness, search endurance and resource/area numbers

Initial POC values are used only to set the initial -values for area nodes, so there is no need to store initial POC values within Washburn objects.

Effectiveness and search endurance values must be accessed by multiple functions during the Washburn allocation process, so these values are stored within Washburn objects and accessed using getEffectiveness() and getSearchEndurance().

Resource and area numbers can be calculated from pointers to resource/area nodes using getNumber(), so there is no need to store these values within the nodes.

### Displaying the Washburn tree

For debugging purposes, it will be very useful to be able to display the current washburn tree. For this reason, the private displayTree() function is included below. The format in which the tree should be displayed is:

* Each node will be displayed over three lines, with the following format:

--> <Resource/Area number> / <type> / <value> / <address>

<parent> / <firstChild> / <nextSibling>

<lagrangeMult> / <eqn>

<Resource/Area number> is printed as an integer (%d).

<type> is displayed as R, A or ROOT, for resources, areas and the root, respectively.

<value> <lagrangeMult> and <eqn> are printed in %.6e format (eg. 3.124721e-5).

<address> is the address of the node, printed in standard pointer format (%p).

<parent>, <firstChild> and <nextSibling> are printed in standard pointer format (%p).

For example, a node may be printed as:

-->2 / R / 1.022348e-6 / 001F2B44

00124EE2 / 001244BB / 002D24B4

4.210404e2 / 1.100238e-1

* The data for each node will be followed by a blank line
* The root node will be left aligned, while all other nodes will be indented 6 spaces further in than their parent.

For example, a section of the tree may appear as:

-->-1 / ROOT / 0.000000e0 / 001FFB24

00000000 / 001F2244 / 00000000

0.000000e0 / 0.000000e0

-->0 / A / 0.000000e0 / 001F2244

001FFB24 / 001F2B44 / 00000000

3.172436e1 / 2.221376e-3

-->2 / R / 1.022348e-6 / 001F2B44

001F2244 / 00000000 / 002D24B4

4.210404e2 / 1.100238e-1

…

…

…

## CONSTANTS

static const double TOLERANCE = sqrt(numeric\_limits<double>::epsilon());

[This is the square root of the machine tolerance value for a double. If numeric\_limits<double> is not defined (as it is supposed to be in C++), obtain the machine precision value from elsewhere.]

static const int ROOTNUMBER = -1;

[This is the resource/area number used for the root node. NULL cannot be used because it is equal to zero, which is a valid resource/area number.]

## PUBLIC FUNCTIONS

NOTE: The Washburn class requires neither a constructor nor a destructor (see §4.1.1).

|  |  |
| --- | --- |
| SIGNATURE | DESCRIPTION |
| double\* calcAllocation(int numResources, int numAreas, const double\* effectiveness, const double\* POC, const double\* searchEndurance) | Uses the Washburn algorithm to calculate the optimal allocation of resources to areas for a single sortie in a Search And Rescue operation.  Arguments are:   * Number of resources (≥ 1). This function sets myNumResources = numResources. * Number of areas (≥1). This function sets myNumAreas = numAreas. * Pointer to first element of 2D array of effectiveness values (double[numResources][numAreas]). Each values is .   This function sets myEffectiveness = effectiveness (for efficiency, no copy of the array is made).   * Pointer to first element of array of (initial) POC values (double[numAreas]). Each value is .   This is passed to createTree() to set the initial lagrangeMult (μ) values for the area nodes.   * Pointer to first element of array of search endurance values (double [numResources]). Each value is . This function sets mySearchEndurance = searchEndurance (for efficiency, no copy of the array is made).   This function must check that all values are within the required ranges (as indicated above).  Return value is:   * Pointer to first element of (dynamically allocated) 1D array of assignment values (double assignments [numResources\*numAreas]). This 1D array represents a 2D table [numResources][numAreas], with assignments[r\*numAreas+a] being the amount of time that resource r is assigned to search area a.   The main part of this function is simply:  createTree(POC);  int resourceNum, areaNum;  while (!isOptimal(resourceNum, areaNum))  { pivot(resourceNum, areaNum); }  double\* val = getSolution();  destroyTree();  return val;  [Note that the Washburn tree is not stored after this function returns, as it is no longer required.]  Based upon Matlab function washburn. |

## PRIVATE FUNCTIONS

|  |  |
| --- | --- |
| SIGNATURE | DESCRIPTION |
| void createTree(const double\* POC) | Creates an initial feasible Washburn tree.  Argument is:   * Pointer to first element of array of (initial) POC values (double[numAreas]).   Begins by allocating memory for the data members myResourceNodes, myAreaNodes and myRoot. It then creates the initial Washburn tree (with all resources fully assigned to the artificial area) by initializing myResourceNodes, myAreaNodes, myRoot and myFalseArea. The initializations include the following:   * myFalseArea = &myAreaNodes[myNumAreas] * myAreaNodes[j].lagrangeMult=POC[j]   (j = 0,…,myNumAreas-1).   * myAreaNodes[myNumAreas].lagrangeMult = sqrt(TOLERANCE) \* exp(-5)   (μ for artificial area).  Based upon Matlab function initialize\_local. |
| void destroyTree(void) | Frees the memory allocated for myArtificialEffectiveness, myResourceNodes, myAreaNodes and myRoot. |
| bool isOptimal(int& resourceNum, int& areaNum) | Returns a boolean value which indicates whether the Karush-Kuhn-Tucker conditions are satisfied.  Also, sets resourceNum and areaNum to values such that  getEffectiveness(resourceNum, areaNum) \* myAreaNodes[areaNum].lagrangeMult / myResourceNodes[resourceNum].lagrangeMult  is maximised.  Based upon Matlab function optimum\_local. |
| void pivot(int resourceNum, int areaNum) | Performs one pivoting operation on the Washburn tree to alter the amount of time for which the resource numbered resourceNum is assigned to search the area numbered areaNum.  Based upon Matlab function pivot\_local (resourceNum and areaNum play the roles of Rnew and Anew in pivot\_local). |
| Washburn::Node\* subtreeRoot(Washburn::Node\* nodePtr) | Recursive function which returns a pointer to the child node of \*myRoot which is the root of a subtree containing \*nodePtr.  Based upon Matlab function subtreeRoot\_local. |
| double sumMu(Washburn::Node\* topPtr) | Recursive function that returns the sum of all n.lagrangeMult values for nodes n where n.type == AREA and either:   * n is within the tree with \*topPtr as root * If \*topPtr is not a child of \*myRoot, n is within a tree that has a following sibling of \*topPtr as root.   If \*topPtr is a child of \*myRoot, returns the sum of n.lagrangeMult values for all area nodes n in the subtree with \*topPtr as root.  Based upon Matlab function sumMu\_local. |
| void createEqn(Washburn::Node\* topPtr, double s) | Recursive function that sets n.eqn = s for all nodes n such that:   * n is within the tree with \*topPtr as root * If \*topPtr is not a child of \*myRoot, n is within a tree that has a following sibling of \*topPtr as root.   If \*topPtr is a child of \*myRoot, sets n.eqn = s for all nodes, n, within the subtree with \*topPtr as root.  Based upon Matlab function createEqn\_local. |
| double solveEqn(Washburn::Node\* topPtr, double tmax, (Washburn::Node\*)& pivotNodePtr) | Recursive function that calculates and returns the value t, which is the result of solving equations for nodes n where:   * n is within the tree with \*topPtr as root * If \*topPtr is not a child of \*myRoot, n is within a tree that has a following sibling of \*topPtr as root.   Also, alters n.eqn for nodes n where:   * n is within the tree with \*topPtr as root * If \*topPtr is not a child of myRoot, n is within a tree that has a following sibling of \*topPtr as root. * n == \*(topPtr -> parent), if topPtr‑>parent is not myRoot.   Also, if tmax, the value returned by applying solveEqn to topPtr->firstChild, and the value returned by applying solveEqn to topPtr‑>nextSibling are all greater than (topPtr->value / topPtr->eqn), after topPtr‑>eqn has been modified within this function, sets pivotNodePtr = topPtr.  [Otherwise, pivotNodePtr is unchanged.]  Based upon Matlab function solveEqn\_local. |
| void updateSubtree(Washburn::Node\* topPtr, double s, double t) | Recursive function that subtracts t\*n.eqn from n.value and multiplies n.lagrangeMult by s for all nodes n where:   * n is within the tree with \*topPtr as root * If \*topPtr is not a child of \*myRoot, n is within a tree that has a following sibling of \*topPtr as root.   If \*topPtr is a child of \*myRoot, subtracts t\*n.eqn from n.value and multiplies n.lagrangeMult by s for all nodes n in the subtree with \*topPtr as root.  Based upon Matlab function updateSubtree\_local. |
| void joinSubtrees(Washburn::Node\* childPtr, Washburn::Node\* parentPtr) | Disconnects \*childPtr from \*(childPtr‑>parent) and makes it the first child of \*parentPtr (the tree with root \*childPtr becomes the first subtree of \*parentPtr).  [NOTE: Before this occurs, childPtr->value should be zero, indicating that there is no assignment for the resource/area combination represented by the nodes \*childPtr and \*(childPtr‑>parent). Check childPtr‑>value, and throw an exception if childPtr‑>value > TOLERANCE.]  Based upon Matlab function joinSubtrees\_local. |
| void rotateSubtree(Washburn::Node\* pivotNodePtr) | Recursive function that rotates the subtree of \*myRoot that contains \*pivotNodePtr so that \*pivotNodePtr becomes the root of this subtree (\*pivotNodePtr becomes the first child of \*myRoot).  This process involves altering n.value for all nodes n in the subtree of \*myRoot that contains \*pivotNodePtr, so that the set of assignments is unchanged.  Based upon Matlab function rotateSubtree\_local. |
| void disconnectSubtree(Washburn::Node\* rootPtr) | Disconnects the tree with \*rootPtr as root from the tree with \*(rootPtr‑>parent) as root by altering the pointer to \*rootPtr from its parent [if \*rootPtr is a first child] or its preceding sibling [if \*rootPtr is not a first child].  Does not alter the pointers to other nodes stored within \*rootPtr.  Based upon Matlab function deleteSubtree\_local. |
| void connectSubtree(Washburn::Node\* childPtr, Washburn::Node\* parentPtr) | Makes \*childPtr the first child of \*parentPtr (the tree with root \*childPtr becomes the first subtree of \*parentPtr) by altering childPtr‑>nextSibling, childPtr‑>parent and parentPtr‑>firstChild.  Does not alter any links to \*childPtr from other nodes.  Based upon Matlab function insertSubtree\_local. |
| double\* getSolution(void) | Dynamically allocates an array:  double assignments[myNumResources\*myNumAreas]  The set of assignments is then extracted from the Washburn tree and stored in this array. (assignments[r\*myNumAreas+a] is the amount of time for which resource r is assigned to search area a.)  &assignments[0] is returned  Although based upon Matlab function getSolution\_local, this function is implemented differently. This is a non‑recursive version that extracts the set of assignments by iterating through the arrays myResourceNodes[0..myNumResources-1] and myAreaNodes[0..myNumAreas-1], rather than traversing the tree. |
| void displayTree(Washburn::Node\* topPtr = myRoot, int indent = 0) | Recursive function that displays the tree with \*topPtr as root, followed by the trees with each of its following siblings as root.  The trees are displayed in the format described in §4.1.5, with \*topPtr, and its following siblings, indented from the left-hand edge of the screen by indent spaces.  If topPtr = myRoot and indent = 0, the entire washburn tree is displayed, with the root node aligned with the left‑hand edge of the screen.  This function need only be included in the class if the compiler tag \_Washburn*\_*testmode is defined. |
| double getSearchEndurance(int resourceNum) | Returns the search endurance of the specified resource (mySearchEndurance[resourceNum]). |
| double getEffectiveness(int resourceNum, int areaNum) | Returns the effectiveness of the specified resource in the specified area.  If the area specified is a real area () then the effectiveness value is myEffectiveness[resourceNum\*myNumAreas+areaNum].  If the area specified is the artificial area (areaNum == myNumAreas) then:   * If myArtificialEffectiveness == NULL, memory is allocated for myArtificialEffectiveness (a single double) and the effectiveness of (all) resources in the artificial area [, where Tk is the search endurance of resource k] is calculated and stored in \*myArtificialEffectiveness. * The value \*myArtificialEffectiveness is returned. |
| int getNumber(Washburn::Node\* nodePtr) | Returns the resource or area number for the node \*nodePtr.  If nodePtr->type == RESOURCE, returns nodePtr – myResourceNodes.  If nodePtr->type == AREA, returns nodePtr-myAreaNodes.  If nodePtr->type == ROOT, returns ROOTNUMBER. |

## DATA MEMBERS

NOTE: None of the data members can be const, as they are not initialized upon construction of a Washburn object.

|  |  |
| --- | --- |
| NAME | DESCRIPTION |
| myNumResources | int: Number of resources. |
| myNumAreas | int: Number of areas (This does NOT include the artificial area added during the allocation process: including this, there are myNumAreas+1 areas.) |
| mySearchEndurance | double\*: Pointer to first element of array [myNumResources] which contains the amount of time for which each resource is available for searching. The actual array is stored external to the Washburn object, and simply accessed via this pointer. |
| myEffectiveness | double\*: Pointer to first element of array [myNumResources][myNumAreas] which contains the effectiveness of each resource in each (real) area. The actual array is stored external to the Washburn object, and simply accessed via this pointer. |
| myArtificialEffectiveness | double\*: Pointer to a (single) double which holds the effectiveness of resources in the artificial area. (All resources have the same effectiveness in the artificial area.) |
| myResourceNodes | Washburn::Node\* (dynamically allocate an array [myNumResources] in createTree()): The nodes in the tree that represent resources. myResourceNodes[r] is the node representing the resource with resource number r. |
| myAreaNodes | Washburn::Node\* (dynamically allocate an array [myNumAreas+1] in createTree()): The nodes in the tree that represent areas. myAreaNodes[a] is the node representing the area with area number a. Note that myAreaNodes[myNumAreas] is the node that represents the artificial area. |
| myRoot | Washburn::Node\*: Pointer to the root node of the tree (which represents neither a resource nor an area). The root node itself is dynamically allocated in createTree(). |
| myFalseArea | Washburn::Node\*: Pointer to the node of the tree which represents the artificial area. myFalseArea = &myAreaNodes[myNumAreas]. |

# CLASS: Washburn::Node

## NOTES

This class is used for data storage only – it does not perform any internal data processing. For this reason, the data members are made public.

For simplicity, the same class is used for Nodes that represent both resources and areas.

## DATA MEMBERS [ALL PUBLIC]

|  |  |
| --- | --- |
| NAME | DESCRIPTION |
| value | double: The amount of time assigned for searching between the area/resource represented by this Node and the resource/area represented by its parent. |
| type | NodeType: RESOURCE, AREA or ROOT |
| parent | Washburn::Node\*: Pointer to parent Node. [NULL if none] |
| firstChild | Washburn::Node\*: Pointer to first child Node. [NULL if none] |
| nextSibling | Washburn::Node\*: Pointer to next sibling Node. [NULL if none] |
| lagrangeMult | double: Lagrange multiplier (μ/λ) for the area/resource represented by this Node. |
| eqn | double: Value used when solving equations during the washburn allocation process. |