SORAL

Library User's Manual

Version 1.3

By: Andre Oboler
12/2/03

Table of Content

Introduction

This manual is aimed at programmers who wish to use the SORAL library in their own applications. If you wish toi develop code for SORAL itself you should read the SORAL Developers Manual (although this document may also be useful). By default SORAL is released under the GNU General Public Licence (GPL). In order to ensure that using SORAL will not adversely effect the copyright status of your own software, please see the current copyright notice of SORAL prior to using it. If there is possibility of a conflict please contact the SARbayes group or Monash University.

The SARbayes project is based at Monash University (Australia), the web site is at: http://sarbayes.org/

What SORAL does for you

SARbayes Optimal Resource Allocation Library (SORAL) is a software library that uses search theory to suggest a theoretically optimal allocation of resources (i.e. search and rescue teams) to areas on a map.

Each area on the map has a probability of containing the missing person or object (the subject). After searching an area the probability of the subject being in that area is reduced, that is we are more confident of them not being there (unless offcourse you found them in which case you should stop the search). As we can never be completely certain we didn't miss subject during the search, these probabilities will approach, but never actually reach zero. A theoretically optimal allocation is one that tries to minimize the search time, that is to assign resources to areas where they will be most effective in lowering the total probability that the subject is in one of the search areas.

Both the resource and area data must be provided to SORAL (see passing data in). SORAL uses this data and one of a number of allocation algorithms (you choose which) to create a set of allocations. An allocation tells you an area to be searched, a resource to search it, and the amount of time (effort) that resource should spend there. Each set of allocations is for a given time slice and SORAL will try and assign all the resource hours. See passing data in for more details, but be aware that you will usually call SORAL repeatedly from your calling application.

Why use SORAL?

SORAL includes the latest in land based search theory. It is developed at Monash University, but includes the latest theory from around the world. The project has the co-operation of leading experts from academia, industry and police search and rescue.

For details of those people and organisations assisting the project please see http://www.sarbayes.org/credit.shtml

SORAL is an ongoing project, but with a stable interface. This means if you build your system to work with SORAL now, you should (with very little work on your part) get improved resource allocation algorithms as SORAL develops in the future. Perhaps best of all, you can build resource allocation into your software with out having to worry about the actual research theory. SORAL encapsulates the allocation theory so all you need to handle are the inputs and resulting allocations.

How to use SORAL

Before using SORAL the parameters for an allocation objects must be created.
After this an allocation object is made. Finally, iterators are used to extract the allocations recommended by SORAL. This process is described below followed by more detail of the input values and a full example.

Passing Data In

Washburn theAllocation(num_res, num_areas, (*effectiveness), availableHours, POC);

num_res	–	An int. This is the total number of resources.
num_areas	-	An int. This is the total number of areas.
effectiveness	-	This is the effectiveness matrix, see below.
availableHours	-	A vector of doubles. The i th member of the vector corresponds to the number of hours available for the i th resource. See vectors below.
POC	-	A vector of doubles. The i th member of the vector corresponds to the probability of containment for the i th resource. See vectors below.

Calling SORAL

To use SORAL, you must create an allocation object. There are various types of allocation objects and each implements a different search algorithm.

Format:
AllocationType nameOfAllocationObject(parameters);

Example:
Washburn theAllocation(num_res, num_areas, (*effectiveness), availableHours, POC);

The actual allocation is carried out during the creation of the allocation object.

You access the result allocations through iterators.

Step one: Set up the Active Areas Iterator.

This iterator will give you the area number of the next area to have resources assigned to it. The area number is an integer.

Example:
```
ActiveAreasIterator* activeAreas;
activeAreas = new ActiveAreasIterator(theAllocation);
```

Step two: Set up some Resource Iterators

A resource iterator will give you a resource number and a time for each allocation in a fixed area. The fixed area is a parameter you pass in when creating it (it is called areaIndex in he code below).

Exmaple:
```
ResourceIterator* activeRes;
activeRes = new ResourceIterator(theAllocation, areaIndex);
```

Step three: Extract an allocation

As each resource iterator will iterate over a list of allocation, you now need to extract the current allocation.

Example:

```
ResourceAssignment* resAssign;
resAssign =activeRes->get();
```

Step four: Access the allocations data

You can work with ResourceAssignment objects in your own code (see example 1), alternatively you can extract the data you need from the assignment, assign it to what ever structure you are using in your program and then delete the resAssign object (see example 2).

Example 1
Simply assign the address of the allocation to some local list of assignments. Note that as the area number is not present your list will need to be area specific.

Example 2
The resource number can be got by: resAssign->getResourceNum();
The amount of time the resource is assigned there can be got by: resAssign ->getTime();
Clean up: delete resAssign

Step five: get the next resource in this area

If this there are any, and repeat steps 3 on. Other wise go to step five.

```
i.e. if  (!(activeAreas->atEnd()))
{
        activeRes++;
}
```

Step six: get the next active area

```
if(!(activeAreas->atEnd()))
{
        ++(*activeAreas);
}
```

Of course all this can be done in a loop. Note also that you need to create the input parameters in the correct formats before you can create any sort of allocation object. The standard parameters all allocation objects take are explain bellow in the section “Passing Data In”.

Getting Information Back

Notes: Description of the iterators and objects. Repeat warning about the SORAL index not necessarily being the same as the users internal index.

Once you have your allocation object you can use it to create iterators. There iterators give you access to allocations or allocation related information. The iterators are:

The iterators

ActiveAreasIterator
This is an iterator over those areas that have something assigned to them. An area that did not get an allocation will be skipped. You can move to the next area with the ++ operator. The area currently pointed to can be extracted with a get command.

Creation example:
```
ActiveAreasIterator * activeAreas;
activeAreas = new ActiveAreasIterator(theAllocation);
```

Increment example:
```
++(* activeAreas);
```

Get value example:
```
activeAreas->get();
```

ResourceIterator
This is an iterator over all resources in a given area. The area is specified at creation and can not be changed. The iterator returns a resource assignment object (of type resAssign) when its get method is called.

Creation example:
```
ResourceIterator * activeRes;
activeRes = new ResourceIterator(theAllocation, areaIndex);
```

Increment example:
```
++(* activeRes);
```

Get value example:
```
activeRes ->get();
```

AreaIterator
This is the complimentry iterator to the ResourceIterator, it takes a resource (fix at creation time) and iterates over all the areas this resource has been assigned to. It returns an object of type areaAssign (an area assignment) when the get method is called.

Creation example:
```
AreaIterator* areaIterator;
areaIterator = new AreaIterator(theAllocation, resourceIndex);
```

Increment example:
```
++(* areaIterator);
```

Get value example:
```
areaIterator ->get();
```

The data members

The active area iterator simply returns an int. This is SORAL's index for the current area.

ResourceAssignment
This item has a time and a resource number. You can access these using the getTime() and getResourceNumber() functions.

AreaAssignment
This item has a time and an area number. You can access these using the getTime() and getAreaNumber() functions.

NOTE:
SORAL's indices are relative to the area / resource position in the vectors and effectiveness matrix, i.e. the first probability in the POC vector will be POC for the area with index 0 if activeArea->get() returned zero it means the area with this POC has the first assignment. Like wise is a zero is returned by the area assignment objects getAreaNumber function.

The Input in more detail

Vectors

POC and availableHours are vectors of doubles. The ordering of areas and resources is not important, but it must be consistent between the vectors and the effectiveness matrix. The position in the vector will also be the index used by the SORAL iterators so you will need to convert this back to your own indices or names when copying the data back… do not mistake the iterators area or resource number as your own internal number (unless your areas and resources happen to be numbered 0 to n and happen to have been passed into SORAL in the correct order).

The effectiveness matrix

The data structure

The effectiveness matrix is of type Array2D. This is a custom data type to SORAL and allows matrix style data access to a dynamically created data structure. The values in the structure are stored as doubles. To create an Array2D you pass in the number of areas and the number of resources.

Example:
```
Array2D* effectiveness = new Array2D(num_areas, num_res);
```

You can access a member of the array as shown below.

Exmaple:
```
effectiveness->value[i][j]=3.14127;
```

The data values

The effectiveness matrix stores effectiveness values for every possible combination of resource and area. An effectiveness value is defined as:

Effectiveness = Effective sweep rate / area size

Effective sweep rate = Effective sweep width for that resource in that area * resource speed for that resource in that area.

As the exact sweep widths and speeds will usually not be available, how you approximate is up to you, some suggestions are provided below in the section missing value.

Missing values

The suggested approaches is to store an average sweep width and speed (taken from all resources under the same standard area conditions) and then "adjust" them for each area. This makes it easier to fill in missing values by extrapolating from known ones. Two methods of doing this are:

A joint correction factor:
Effective sweep rate = Effective sweep width * resource speed * correction factor

Two separate correction factors:
Effective sweep rate = (Effective sweep width * correction factor1) * (resource speed * correction factor2)

The creation of useful effectiveness values is an ongoing research task. SARBayes will hopefully develop methods and tools for dealing with this problem in the future.

A Walk Through

```
#include <iostream>
#include "Array2D.h"
#include <vector>
#include "Allocatn.h"
#include "containr.h"

using namespace std;

// function definitions
void printAssignments(Allocation& theAllocation);

int main(int argc, int argv[])
{
	/* ** Set up the input values ** */
	int num_areas=2;
	int num_res=1;
	Array2D* effectiveness = new Array2D(num_areas, num_res);
	vector<double> availableHours;
	vector<double> POC;

	// int Area, int Resource = double effectiveness
	effectiveness->value[0][0]=2;
	effectiveness->value[1][0]=2;


	// Add each resource's time available to the availableHours Vector
	availableHours.push_back(5);


	// Add each areas probability to the POC vector
	POC.push_back(20);
	POC.push_back(40);

	/* ** create the allocation object ** */
Washburn theAllocation(num_res, num_areas, (*effectiveness), availableHours, POC);

	// Now display results

	/* ** use iterators to extract the data (see next page) ** */
	printAssignments(theAllocation);

	return 1;
}
```

```
/* ** Use iterators to extract the data ** */
void printAssignments(Allocation& theAllocation)
{
	ActiveAreasIterator* activeAreas;
	int areaIndex, resourceIndex;
	ResourceIterator* activeRes;
	ResourceAssignment* resAssign;
	double time;

	activeAreas= new ActiveAreasIterator(theAllocation);
	areaIndex=activeAreas->get();
	activeRes= new ResourceIterator(theAllocation, areaIndex);
	resAssign=activeRes->get();
	resourceIndex=resAssign->getResourceNum();
	time=resAssign->getTime();

	while(!(activeAreas->atEnd()))
	{
		cout << "Area: " << areaIndex << "  Resource: " << resourceIndex << "
		Time: " << time << "\n";
		// if this area still has more assignments
		++(*activeRes); //incriment
		if (!(activeRes->atEnd()))
		{
			resAssign=activeRes->get();
			resourceIndex=resAssign->getResourceNum();
			time=resAssign->getTime();
		}
		else
		{
			++(*activeAreas); //incriment
			if (!(activeAreas->atEnd()))
			{
				areaIndex=activeAreas->get();
				delete activeRes;
				activeRes= new ResourceIterator(theAllocation,
				areaIndex);
				resAssign=activeRes->get();
				resourceIndex=resAssign->getResourceNum();
				time=resAssign->getTime();
			}
		}
	}
	delete activeRes;
	delete activeAreas;
}
```

How to use the information you get back

Standard usage

Using the iterators as suggested in the calling SORAL section will allow you to extract all allocations from SORAL. You might also like to use the iterators to:

Build a list of all resources that a particular area has assigned to it (for example to display in a GUI object)

Build a list of all areas a particular resource visits (for example to display in a GUI object)

Build a list of areas being search in this round

Build a list of areas not being search in this round

Research and evaluation uses

Create more than one allocation objects so you can compare different algorithms suggestions

Use the same allocation type but vary the resources to see how it might effect the search (or to gauge how effective a particular search and rescue unit would be for searches in your region)

SAR Management uses

Keep a list of allocation objects so you can view where resources were assigned over time.

Keep a list of allocation objects so you can view all resources that entered a given area (useful is a contamination risk is later discovered, or evidence in a crime scene is found to have been tampered with).

Allocation Algorithm Details

Each of the algorithms is built to cope with specific limitations. More algorithms are currently being built and will be released with the next version of the SORAL library. The three currently implemented algorithms are: Charnes Cooper, Washburn and UserDef. The purpose of each of these is described below.

Name: Charnes-Cooper

This algorithm assumes only one resource. You can achieve this by lumping all your ground searchers together. For example if you had 60 searchers for 6 hours each, you would list your available hours as 360 hours for a single “ground searcher” resource. The algorithm assumes it can infinitely split resources.

Note for search Managers / implementation:
For this to work your ground searchers should have similar effectiveness… this can be achieved by having less effective searchers so slower and for longer, but without including this extra time in your available hours. For example if one team of 6 searchers (in the example above) was half as effective, we would ask them to pace themselves at half speed in order to do a more “thorough” job. Our total hours for the search algorithm would be 54*6 + 6*3 = 342 rather than 360.

Name: Washburn

This algorithm accepts many resources. Where only one resource is given it will become the Charnes Cooper algorithm. Resources should still be grouped into similar types with similar effectiveness values. The algorithm assumes it can infinitely split resources.

Name: UserDef

This is not an automated allocation class, but rather one that allows you to edit and create allocations manually. Any other type of allocation class can be converted to a UserDef class.

The following continues the example above and can be insert once the Washburn object is created.

```
UserDef ChangedAllocation(theAllocation);
ChangedAllocation.remove(0,0,0.5);
printAssignments(ChangedAllocation);
```

UserDef allows allocations to deleted and created as well as altered (shifting time shifted between allocation). Note that UserDef does do some booking and will store all “removed” resource hours. You can add these to an area with the add function (specifying a resource and area but not a an amount of time). To ensure no resource time is lost you may like to assign each resource to one of the areas before exiting from the code that alters UserDef. If all resource hours have been assigned somewhere, this code will have no effect. If not this code will prevent a resource leak.

Assumptions:

Infinitely split resources: the allocations are of a precision that is theoretical but probably not practical. You may with to round search time to something more practical