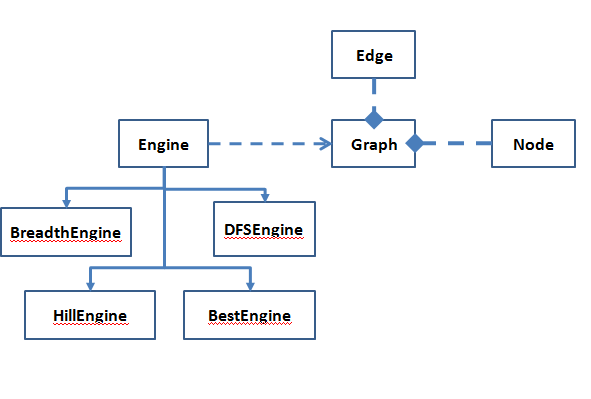
**Software Architecture**

All the algorithms (for both Questions 1 and 2) use a class, called *Graph*, which allows the algorithms (Bread First Search, DFS, Best First Search, Hill Climbing) to create a graph and visit and act on each node as necessary.

Below diagram shows a high-level architecture for the software



**Breadth First Search**

**Breadth First Search Overview**

In Breadth first search, all the nodes at depth d in the search tree are expanded before the nodes at depth d + 1. That is, shallowest node is expanded first. The solution may not always be the least-cost solution for a general path cost function. In this search, fringe is a FIFO queue( new successors go at end)

Properties of BFS

* Complete: Yes (if maximum branching factor of the search tree is finite). If there is a solution, breadth-first search is guaranteed to find it, and if there are several solutions, breadth-first search will always find the shallowest goal state first
* Time: 1+b+b2+b3+… +bd + b(bd-1) = O(bd+1), where b=branching factor of the states (and of the search tree), d=path length of the solution
* Space: O(bd+1) . Keeps every node in the memory
* Optimality: Yes, if cost = 1 per step

The memory requirements (space) are a bigger problem for breadth-first search than the execution time

**Algorithm Description**

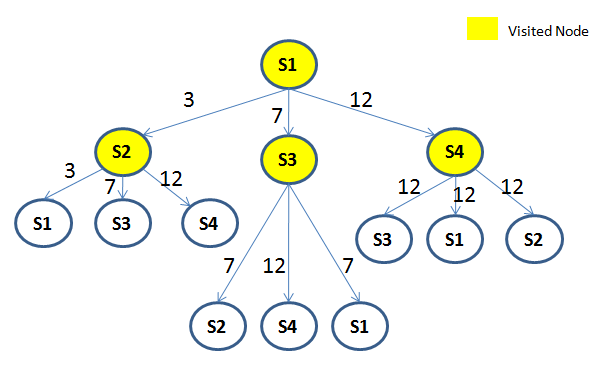
This section explains the generic high-level BFS algorithm to solve Sarawak Bridge Problem

1. Expand parent node A to determine if parent node has children
2. If parent has children, then
   1. Expand child node B
   2. Move parent A and child B to the other side of the bridge
   3. Move parent Node A back

Above steps are repeated until all four people have crossed the bridge

Basic concept is that a parent node helps all its child nodes to cross the bridge. For example, parent node A will carry its child node B to other side of the bridge and then cross the bridge again to carry its child node C. This process will continue until all four people have crossed the bridge

**How nodes are expanded in Breadth First Search**



**Pseudocode**

currentNode = rootNode; //Root node (rootNode=s1)

Edge currentEdge=null; //Edge from root node to one of its child nodes (e.g. from s1 to s2)

totalTimeLimit=20 //Time limit entered by user

fringeEdgeList //List of unvisited edges in the graph

do

{

If ( goal state has been reached within totalTimeLimit specified by user)

{

If currentNode has not been visited before, Add currentNode to visitedNodeList

goalReached=true;

break;

}

Else if node has not been visited before

{

Add currentNode to visitedNodeList

If currentNode has children AND path cost to reach from parent to child < = timeLimit entered by user

{

edgeList = Expand currentNode to get a list of Edges from currentNode to child nodes. Remove all edges that have already been visited and/or already exist in fringeEdgeList

Append this list of refined edges to fringeEdgeList

If parent node (e.g. S1) of a child node (e.g. S2) has already crossed the bridge, then

{

If the path cost of parent node (S1) crossing back <= totalTimeLimit entered by user, then

{

bring the parent (e.g. S1) back

if path cost of parent & child nodes (e.g. S1 & S2) crossing the bridge together <= totalTimeLimit entered by user, then

{

Make parent (e.g. S1) and child (e.g. S2) cross the bridge together

updateTotalTimeTaken();

}

}

}

Else if (parent node (e.g. S1) and child node (e.g. S2) are on the same side, then

{

if path cost of parent & child nodes (e.g. S1 & S2) crossing the bridge together <= totalTimeLimit entered by user, then

{

Make parent (e.g. S1) and child (e.g. S2) cross the bridge

updateTotalTimeTaken();

}

}

}

}

If there are more fringe (unvisited) edges in fringeEdgeList, then

{

currentEdge=next edge in fringeEdgeList;

currentNode= currentEdge.getChild()

}

}while (currentEdge != null && currentNode != null);

**Depth First Search**

**Overview**

DFS always expands on deepest unexpanded node. Only when the search hits a dead end (a nongoal node with no expansion) does the search go back and expand nodes at shallower levels.

Properties

1. Complete: No.Fails in infinite-depth spaces & spaces with Loops. This search is complete in finite spaces
2. Time Complexity: O(bm): terrible if m ( maximum depth of the state space) is much larger than d (depth of the least-cost solution). if solutions are dense, DFS may be much faster than BFS
3. Space: O(bm), i.e., linear space
4. Optimal: No

**Algorithm Description**

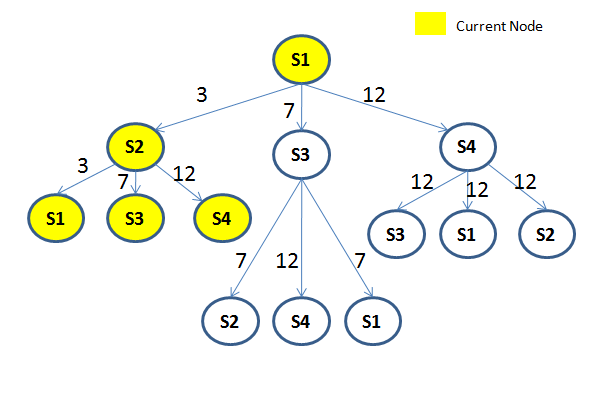
This section explains the generic high-level DFS algorithm to solve Sarawak Bridge Problem

1. Expand parent node A to determine if parent has children
2. If parent has children, then
   1. Determine if first child node *B* has its own children
   2. Move parent A and child B to the other side of the bridge
   3. Move child B back

Above steps are repeated until all four people have crossed the bridge

Basic concept is that a child node helps its parent node to cross the bridge. Child node drops its parent on other side of the bridge and then crosses back. For example, child node B will carry its parent node A to the other side of the bridge. If child node B itself has children, then , child node B will cross the bridge back. This process will continue until all four people have crossed the bridge

**How nodes are expanded in DFS**



**Pseudocode**

currentNode = rootNode;

Edge currentEdge=null; //Edge from root node to one of its child nodes (e.g. from s1 to s2)

totalTimeLimit=0 //Time limit entered by user

fringeEdgeList //List of unvisited edges in the graph

do

{

If goal state has been reached within timeLimit specified by user

{

break;

}

Else if currentNode has not yet been visited

{

Add currentNode to list of visited nodes

If currentNode has children AND path cost to reach from parent to child < = timeLimit entered by user

{

edgeList = Expand currentNode to get a list of Edges from currentNode to child nodes. Remove all edges that have already been visited and/or already exist in fringeEdgeList

for each edge in edgeList

{

if(edge has children)

{

Node edgeChild=edge.getCchild(i);

If child of this edge has not already been visited, Add this edge at index 0 of fringeEdgeList

}

}

}

If a parent node of currentNode is on the other side of the bridge with the torch light, then

{

Move the parent node across the bridge, if pathCost to move the parent < = timeLimit entered by user

Move parent Node and child node together, if pathCost to move both the nodes <= timeLimit entered by user

}

If there are more fringe (unvisited) edges in fringeEdgeList, then

{

currentEdge=next edge in fringeEdgeList;

currentNode= currentEdge.getChild()

}

else

{

currentEdge=null;

currentNode=null;

}

} while (currentEdge != null && currentNode != null);

**Depth First Search vs Breadth First Search**

**Implementation differences:**

Breadth first search has been implemented using First In First Out (FIFO) queue, where successors go at the end of the queue. Depth First Search has been implemented using Last In First Out (LIFO) queue, where successors are put in the front of the queue

**Preferred Algorithm:** From the above, it can be concluded that for the Sarawak Bridge problem, Depth First Search would be a better algorithm. DFS can find out the solution by expanding & checking fewer number of nodes, compared to Breadth First Search

**Best First Search**

**Overview**

In Best First Search, node is selected for expansion based on anevaluation function f(n). The evaluation function measures distance to the goal and selects the node which appears best. In Best first Search, fringe is a queue sorted in decreasing order of desirability

Special cases of BFS: greedy search, A\* search

**Algorithm Description**

This section explains the generic high-level BFS algorithm to solve Sarawak Transport Problem

1. Expand parent node to determine if it has children
2. If parent has children, then, determine the best child using evaluation function f*=g+h*
3. Keep repeating steps 1,2 until a solution is found

**Pseudocode**

Node currentNode = rootNode; //variable holding reference to currentNode

Edge currentEdge = null; //variable holding reference to currentEdge of graph

fringeEdgeList //List of unvisited edges in the graph

do

{

ChecGoalState(). If goal state has been reached, then

{

Add current node to list of visitedNodes

break;

}

Else if currentNode has not already been visited, then

{

Add current node to list of visitedNodes

If currentNode has children

{

edgeList = Get list of all edges from currentNode to child nodes. Refine the list to remove all edges that have already been visited or already exist in fringeEdgeList

if(edgeList.size()>0)

{

bestEdge = findBestEdge(edgeList), using evaluation function f=g+h;

bestChildNode = bestEdge.getChild();

Insert all fringe edges (except bestEdge) to the beginning of fringeEdgeList

Next, insert bestEdge at the beginning of fringeEdgeList

}

else { backTrack() }

}

else {backTrack () }

}

If there are more fringe (unvisited) edges in fringeEdgeList, then

{

currentEdge=next edge in fringeEdgeList;

currentNode= currentEdge.getChild()

}

else

{

currentEdge=null;

currentNode=null;

}

} while (currentEdge != null && currentNode != null);

**Hill Climbing Search**

**Overview**

Looks one step ahead to determine if any successor is better than the current state; if there is, move to the best successor. Does not allow backtracking or jumping to an alternative path since it does not remember where it has been.

Problems with hill climbing

* Local Maxima: peaks that aren’t the highest point in the space
* Plateaus: the space has a broad flat region that gives the search algorithm no direction (random walk)

**Algorithm Description**

This section explains the generic high-level Hill Climbing algorithm to solve Sarawak Transport Problem

1. Expand parent node to determine if it has children
2. If parent has children, then, determine the best child using evaluation function *h*
3. Keep repeating steps 1,2, until a solution is found or local maxima is reached

**Pseudocode**

Node currentNode = rootNode; //variable holding reference to currentNode

Edge currentEdge = null; //variable holding reference to currentEdge of graph

fringeEdgeList //List of unvisited edges in the graph

do

{

ChecGoalState(). If goal state has been reached, then

{

Add current node to list of visitedNodes

break;

}

Else if currentNode has not already been visited, then

{

Add current node to list of visitedNodes

If currentNode has children

{

edgeList = Get list of all edges from currentNode to child nodes. Refine the list to remove all edges that have already been visited or already exist in fringeEdgeList

if(edgeList.size()>0)

{

bestEdge = findBestEdge(edgeList), using evaluation function f=h;

bestChildNode = bestEdge.getChild();

Insert all fringe edges (except bestEdge) to the beginning of fringeEdgeList

Next, insert bestEdge at the beginning of fringeEdgeList

}

else { search failed }

}

else {search failed }

}

If there are more fringe (unvisited) edges in fringeEdgeList, then

{

currentEdge=next edge in fringeEdgeList;

currentNode= currentEdge.getChild()

}

else

{

currentEdge=null;

currentNode=null;

}

} while (currentEdge != null && currentNode != null);

**Best First Search vs Hill Climbing Search**

**Implementation Difference:**

1. Best First Search was implemented using evaluation function *f=g+h.* Hill Climbing Search was implemented using evaluation function *f=h*
2. Best First Search can backtrack, whereas Hill Climbing search cannot. Hill Climbing Search terminates at local maxima
3. In Best First Search, a LinkedList is used to keep a track of visited edges

**Preferred Algorithm:** For Sarawak Road Transportation Problem, given heuristics are inadmissible. Hence, only Best First Search will yield a result, as it can backtrack. Hill Climbing gets stuck at Bau. Thus, Best First Search is the preferred algorithm

**A\* Search**

**Overview**

Best-first search using *f* as the evaluation function and an admissible h function is known as A\* search A\* Search is the best known form of Best-First Search. A\* Search avoids expanding paths that are already expensive.

Evaluation function f(n)=g(n) + h(n), where

* g(n) the cost (so far) to reach the node
* h(n) estimated cost to get from the node to the goal
* f(n) estimated total cost of path through n to goal

A\* search uses an admissible heuristic. A heuristic is admissible if it never overestimates the cost to reach the goal.

Properties

1. Complete: Yes (as long as the memory supports the depth and branching factor of the tree)
2. Optimal: Yes (if heuristic is admissible)
3. Time & Space Complexity: O(bd) . Since A\* must keep track of the nodes evaluated so far (and also the discovered nodes to be evaluated),

**Algorithm Description**

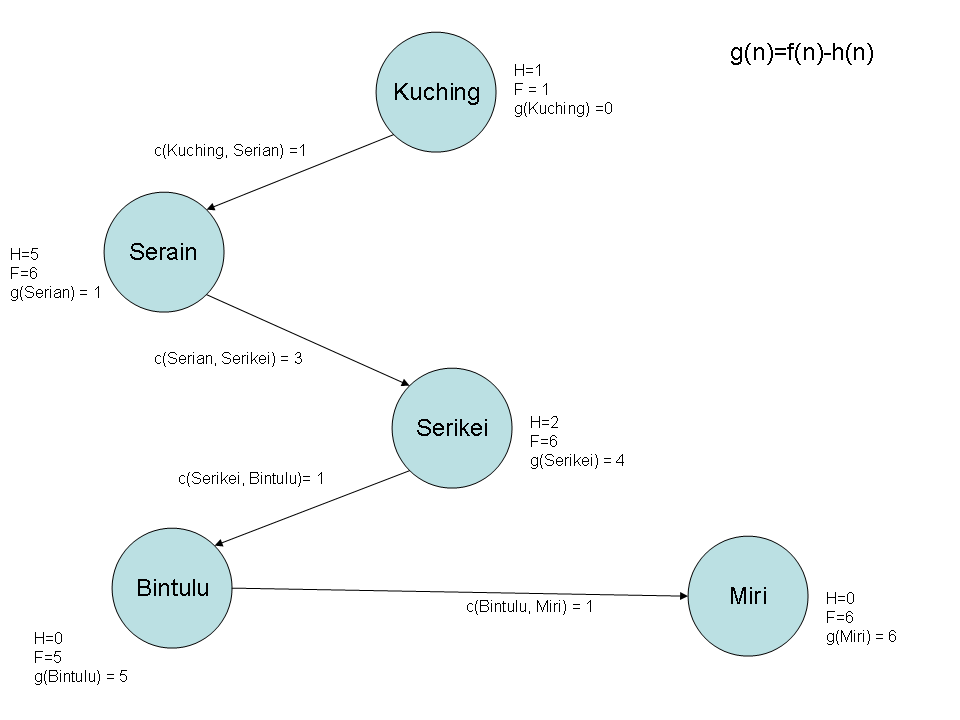
This section explains the generic high-level A\* algorithm to solve Sarawak Transport Problem

1. Expand parent node
2. Find out if parent has children
3. If parent has children, then, determine the best child node using evaluation function *g(n)=f(n)-h(n).*
4. Keep repeating steps 1,2,3 until a solution is found

**Optimal Solution**

This section explains how optimal solution was achieved

|  |
| --- |
| Select Kuching |
| Find *g(Kuching)* using evaluation function  *g(n)=f(n) - h(n)*  g(Kuching) = f(Kuching) – h(Kuching) = 1-1=0  c(Kuching, Kuching) = 0 |
| checkGoal(Kuching). Since Kuching is not the goal state, proceed |
| Expand Kuching to get its child nodes  New child nodes: Sibu, Serian |
| Evaulate each child node using evaluation function *g(n)=f(n) - h(n)*  g(Sibu)=f(Sibu)-h(Sibu)=5-3=2  c(Kuching,Sibu) = 2  g(Serian)=f(Serian)-h(Serian)=6-5=1  c(Kuching, Serian) =1 |
| Since g(Serian) < g(Sibu)  Select Serian |
| CheckGoal(Serian). Since Serian is not the goal state, proceed |
| Expand Serian to get its child nodes  New child nodes: Sarikei |
| Evaulate each child node using evaluation function *g(n)=f(n) - h(n)*  g(Sarikei)=f(Sarikei) – h(Sarikei) = 6-2=4  c(Serian, Serikei) =3 |
| Select Serikei |
| CheckGoal(Serikei). Since Serikei is not goal state, proceed |
| Expand Serikei to get child nodes  New Child nodes: Bintulu, Miri |
| Evaulate each child node using evaluation function *g(n)=f(n) - h(n)*  g(Bintulu)=f(Bintulu)-h(Bintulu) =5-0=5  c(Serikei, Bintulu) = 1  g(Miri)=f(Miri)-h(Miri)=7-0=7  c(Serikei, Miri) = 3 |
| Since g(Bintulu) < g(Miri)  Select Bintulu |
| CheckGoal (Bintulu). Since Bintulu is not the goal state, proceed |
| Expand Bintulu to get child nodes  New child nodes: Miri |
| Evaulate each child node using evaluation function *g(n)=f(n) - h(n)*  g(Miri)=f(Miri) – h(Miri) = 6-0=6  c(Bintulu, Miri) = 1 |
| Select Miri |
| CheckGoal (Miri) . Goal state has been reached |



**Flow Chart with Transition Costs for all edges**

