# Search

**Depth-First Search**

1. Set **N** to be a list of initial nodes.
2. If **N** is empty, then exit and signal failure.
3. Set **n** to be the first node in **N**, and remove **n** from **N**.
4. If **n** is a goal node, then exit and signal success.
5. Otherwise, add the children of **n** to the front of **N** and return to step 2.

# Search (cont)

**Breadth-First Search**

1. Set **N** to be a list of initial nodes.
2. If **N** is empty, then exit and signal failure.
3. Set **n** to be the first node in **N**, and remove **n** from **N**.
4. If **n** is a goal node, then exit and signal success.
5. Otherwise, add the children of **n** to the end of **N** and return to step 2.

# Search (cont)

**Depth Limited Search**

1. Set **N** to be a list of initial nodes.
2. If **N** is empty, then exit and signal failure.
3. Set **n** to be the first node in **N**, and remove **n** from **N**.
4. If **n** is a goal node, then exit and signal success.
5. If the depth of **n** is equal to **MAX**, go to step 2.
6. Otherwise, add the children of **n** to the front of **N** and return to step 2.

# Search (cont)

**Iterative-Deepening Search**

1. Set **N** to be a list of initial nodes. Set **MAX** to 1.
2. Perform Depth-Limited Search
3. If success, exit and signal success
4. Otherwise, set **MAX** to **MAX** $+$ 1, and go to step 2.

# GeneralQueue.java

```
public interface GeneralQueue {
/** Add a new object to the queue.*/
   public void add(SearchNode object);

/** Remove the next object in queue order.*/
   public SearchNode removeFront();
/** Predicate to determine if queue is empty.*/
   public boolean isEmpty();
}
```

# LIFOQueue.java

```
import java.lang.*;
import java.util.*;
public class LIFOQueue
     implements GeneralQueue {
   Stack<SearchNode> stack;
   public LIFOQueue() {
        stack = new Stack<SearchNode>();}
   public void add(SearchNode object) {
      stack.push(object);}
   public SearchNode removeFront() {
      return stack.pop(); }
   public boolean isEmpty() {
      return stack.empty();}
}
```

# FIFOQueue.java

```
import java.lang.*;
import java.util.*;
public class FIFOQueue implements GeneralQueue {
   protected ArrayList<SearchNode> fifo;
   public FIFOQueue() {
       fifo = new ArrayList<SearchNode>();}
   public void add(SearchNode object) {
      fifo.add(object);}
   public SearchNode removeFront() {
      SearchNode object= fifo.get(0);
      fifo.remove(0);
      return object;}
   public boolean isEmpty() {
      return fifo.size() == 0;}
}
```

# State.java

```
import java.lang.*;
import java.util.*;
/** Interface for any object that can be
   manipulated as a state
   by an implementation of SearchMethod.  */
public interface State {

/**
   Returns whether this state is a goal node.*/
   public boolean isGoal();
/** Returns a collection  of
     successors of the state.*/
   public ArrayList successors();
}
```

# SearchNode.java

```
import java.lang.*;
import java.util.*;
/** Search node rep node in a search tree,
supplies needed funcs for implementing a search.*/
public class SearchNode {
/** State at this node */
   protected State state;
/** Reference back to parent node. */
   protected SearchNode parent;
/** Operation that was applied to parent */
   protected String appliedOp;
/** Depth of this node */
   protected int depth;
/** Cost of getting to this node */
   protected float pathCost;
```

# SearchNode.java (cont)

```
/** No-argument constructor needed
for newInstance() */
   protected SearchNode() { }
/** Constructor takes a state and makes it
a parentless search node */
   public SearchNode(State startState) {
      state = startState;
      parent = null;
      appliedOp = null;
      depth = 0;
      pathCost = 0;}
```

# SearchNode.java (cont)

```
/** Returns state of this node. */
   public State getState() {
         return state;}
/** Returns parent of this node. */
   public SearchNode getParent() {
         return parent;}
/** Returns applied operation for this node. */
   public String getAppliedOp() {
         return appliedOp;}
/** Returns depth of this node. */
   public int getDepth() {
         return depth;}
```

# SearchNode.java (cont)

```
/** Returns cost of getting to this node */
   public float getPathCost() {
        return pathCost;
   }
/** Expands a node into its successors */
public void expand(GeneralQueue expandInto) {
ArrayList<Successor> successorList =
        getState().successors();
for (Successor next: successorList)
   expandInto.add(makeNode(next) );
  }
}
```

# SearchNode.java (cont)

```
/** Makes a new node of the same type as this one,
using a successor */
public SearchNode makeNode(Successor successor) {
SearchNode newNode;
try {
 newNode = (SearchNode) getClass().newInstance();
 newNode.state = successor.getState();
 newNode.parent = this;
 newNode.appliedOp = successor.getOperatorName();
 newNode.depth = depth+1;
 newNode.pathCost = pathCost + successor.getCost();
  return newNode;
  } catch (InstantiationException e) {
  } catch (IllegalAccessException e) {}
  return null; } }
```

# SearchMethod.java

```
import java.lang.*;
import java.util.*;
/**
   Interface for any object that
     implements a search algorithm.
*/
public interface SearchMethod {
/**
   Perform the search.
*/
   public SearchNode search();
}
```

# Successor.java

```
import java.lang.*;
import java.util.*;
/**
   Bundles together a new state,
the name of the operator used to
   get there, and the cost of the operation.*/
public final class Successor {
/** Successor State */
  protected State state;
/** Operation to reach successor */
  protected String operatorName;
/** Cost of operation */
  protected float cost;
```

# Successor.java (cont)

```
/** Constructor sets all values of
successor object.   */
  public Successor
       (State state, String operatorName,
                    float cost) {
     this.state = state;
     this.operatorName = operatorName;
     this.cost = cost;}
/**
  Returns string describing operation. */
  public String getOperatorName() {
    return operatorName;}
```

# Successor.java (cont)

```
/**
  Returns cost of performing operation.*/
  public float getCost() {
    return cost;
  }
/**
  Returns new state.
*/
  public State getState() {
    return state;
  }
}
```

# BreadthFirstSearch.java

```
import java.lang.*;
import java.util.*;
/**
     Breadth-first search:
redefines constructor to use FIFO queue.
*/
public class BreadthFirstSearch
        extends GeneralQueueSearch {

 public BreadthFirstSearch(State startState) {
  super(new SearchNode(startState),
                           new FIFOQueue());
    }
}
```

# DepthFirstSearch.java

```
import java.lang.*;
import java.util.*;
/**
     Depth-first search:
redefines constructor to use stack.
*/
public class DepthFirstSearch
            extends GeneralQueueSearch {
 public DepthFirstSearch(State startState) {
   super(new SearchNode(startState),
                        new LIFOQueue());
    }
}
```

# DepthBoundedSearch.java

```
import java.lang.*;
import java.util.*;
public class DepthBoundedSearch
           implements SearchMethod {
/**Stack for depth-first search.*/
   protected LIFOQueue Q;
/** Bound on depth.*/
   int maxDepth;
/** Constructor takes initial state & depth bound.*
public DepthBoundedSearch(State startState,
                                int maxDepth) {
    Q = new LIFOQueue();
    Q.add( new SearchNode(startState) );
    this.maxDepth = maxDepth;}
```

# DepthBoundedSearch.java (cont)

```
/** Performs depth-bounded search
       from initial state.*/
 public SearchNode search() {
   while (!Q.isEmpty()) {
    SearchNode expandNode =
             Q.removeFront();
     if (expandNode.getState().isGoal()) {
       return expandNode;
     } else if (expandNode.getDepth()
                        < maxDepth) {
         expandNode.expand(Q);
        }
     }
     return null;
   } }
```

# IteratedDeepeningSearch.java

```
import java.lang.*;
import java.util.*;
/** This class implements an
   iterative deepening search. */
public class IteratedDeepeningSearch
                implements SearchMethod {
  /** Start state, which must be stored
       to implement repeated searches. */
   State startState;
  /* Constructor takes starting state */
   public
    IteratedDeepeningSearch(State startState) {
       this.startState = startState;
   }
```

# IteratedDeepeningSearch.java (cont)

```
/* Implementation of iterative
       deepening search. */
   public SearchNode search() {
 for (int depth=1 ; ; depth++) {
 SearchNode node
    = (new DepthBoundedSearch
         (startState, depth)).search();
      if (node != null) return node;
       }
   }
}
```

# CLASSPATH

- All Search Code included in `Search.jar`
- Must include in Class Path
- On CSlab, put the following line in your .profile

```
export CLASSPATH=Search.jar:.
```

# Traversable.java

```
import java.lang.*;
import java.util.*;
/** Interface for a state space that
can be traversed by applying operators.*/
public interface Traversable {
/** Return state obtained by applying op.
  null if op is not valid here. */
   public State applyOperator(String op);
/** Return cost of applying op. */
   public float costOf(String op);
/** Get all operators valid from this state. */
   public ArrayList<String> validOperators();
}
```

# TraversableState.java

```
import java.lang.*; import java.util.*;
/** A state class that gives a general
implementation of successors() function for any
traversable state space.*/
public abstract class TraversableState
     implements State, Traversable {
/** Return successors using methods in
    Traversable interface.*/
   public ArrayList<Sucessor> successors() {
       ArrayList successorList = new ArrayList();
       ArrayList<String> opList = validOperators();
       for (String op: oplist)
          successorList.add(new Successor
              (applyOperator(op), op, costOf(op)) )
       return successorList;}}
```

# TwoThreeState.java

```
import java.lang.*;
import java.util.*;
public class TwoThreeState
    extends TraversableState implements Heuristic {
   int stateValue;
   public TwoThreeState() {
       stateValue = 0; }
   public State applyOperator(String op) {
       TwoThreeState nextState = new
                                 TwoThreeState();
       if (op.equals("add2")) {
           nextState.stateValue= stateValue+2;
       } else if (op.equals("add3")) {
           nextState.stateValue= stateValue+3;}
       return nextState;}
```

```
public float costOf(String op) {
    if (op.equals("add2")) {
       return 2;
    } else {
       return (float) 4;
    }
}
```

# TwoThreeState.java (cont)

```
    public Collection validOperators() {
        ArrayList opList = new ArrayList();
        opList.add("add3");
        opList.add("add2");
        return opList; }
    public boolean isGoal() {
      return (stateValue > 0
           && stateValue%23==0) ;}
    public float h() {
        float hVal = 23 - stateValue;
        if (hVal<0) return 0;
        else return hVal;}
    public String toString() {
        return "(" + stateValue + ")";}
}
```

# TestSearch.java

```
public class TestSearch {
public static void main(String argv[]) {
System.out.println
("Trivial search space based on adding 2 or 3");
System.out.println("DFS:");
listPath(
  ( new DepthFirstSearch
     (new TwoThreeState()) ).search() );
System.out.println();
System.out.println("Depth Bounded (depth 7):");
listPath(
   ( new DepthBoundedSearch
     (new TwoThreeState(),7) ).search() );
System.out.println();
System.out.println("BFS:");}}
```

# TestSearch.java(cont)

```
protected static void listPath(SearchNode node) {
  if (node == null) {
    System.out.println("No solution");
      return;}
    while (node.getParent()!=null) {
    System.out.println( "State: " +
                 node.getState() +
      " Depth: " + node.getDepth() +
      " Cost: " + node.getPathCost() +
      " by applying " + node.getAppliedOp()
       );
          node = node.getParent();
       }
```

# TestSearch.java(cont)

```
System.out.println
 ( "Starting at state: " +
               node.getState());}  }
```

# Output

```
State: (46) Depth: 23 Cost: 46.0 by applying add2
State: (44) Depth: 22 Cost: 44.0 by applying add2
State: (42) Depth: 21 Cost: 42.0 by applying add2
State: (40) Depth: 20 Cost: 40.0 by applying add2
State: (38) Depth: 19 Cost: 38.0 by applying add2
State: (36) Depth: 18 Cost: 36.0 by applying add2
State: (34) Depth: 17 Cost: 34.0 by applying add2
State: (32) Depth: 16 Cost: 32.0 by applying add2
State: (30) Depth: 15 Cost: 30.0 by applying add2
State: (28) Depth: 14 Cost: 28.0 by applying add2
State: (26) Depth: 13 Cost: 26.0 by applying add2
State: (24) Depth: 12 Cost: 24.0 by applying add2
State: (22) Depth: 11 Cost: 22.0 by applying add2
State: (20) Depth: 10 Cost: 20.0 by applying add2
```

# Output

```
State: (18) Depth: 9 Cost: 18.0 by applying add2
State: (16) Depth: 8 Cost: 16.0 by applying add2
State: (14) Depth: 7 Cost: 14.0 by applying add2
State: (12) Depth: 6 Cost: 12.0 by applying add2
State: (10) Depth: 5 Cost: 10.0 by applying add2
State: (8) Depth: 4 Cost: 8.0 by applying add2
State: (6) Depth: 3 Cost: 6.0 by applying add2
State: (4) Depth: 2 Cost: 4.0 by applying add2
State: (2) Depth: 1 Cost: 2.0 by applying add2
Starting at state: (0)
```

# Output

```
Depth Bounded (depth 7):
No solution
BFS:
State: (23) Depth: 8 Cost: 30.0 by applying add2
State: (21) Depth: 7 Cost: 28.0 by applying add3
State: (18) Depth: 6 Cost: 24.0 by applying add3
State: (15) Depth: 5 Cost: 20.0 by applying add3
State: (12) Depth: 4 Cost: 16.0 by applying add3
State: (9) Depth: 3 Cost: 12.0 by applying add3
State: (6) Depth: 2 Cost: 8.0 by applying add3
State: (3) Depth: 1 Cost: 4.0 by applying add3
Starting at state: (0)
```

# Output (cont)

```
Iterated Deepening Search:
State: (23) Depth: 8 Cost: 30.0 by applying add3
State: (20) Depth: 7 Cost: 26.0 by applying add3
State: (17) Depth: 6 Cost: 22.0 by applying add3
State: (14) Depth: 5 Cost: 18.0 by applying add3
State: (11) Depth: 4 Cost: 14.0 by applying add3
State: (8) Depth: 3 Cost: 10.0 by applying add3
State: (5) Depth: 2 Cost: 6.0 by applying add3
State: (2) Depth: 1 Cost: 2.0 by applying add2
Starting at state: (0)
logic% exit
```

# Application

**Scheduling Astronomical Observations**

- CERES – a real-time scheduling system for astronomical observations.
- Developed at NASA Ames Research Center.
- CERES combines scheduling of requested observations with the control of the telescopes and can dynamically respond and reschedule in the event that conditions make an observation impossible.
- Reduces the support staff and operations costs.
- Provides improved utilization of telescopes and increased flexibility.

# Application

**Scheduling Space Shuttle Maintenance**

- GPSS – Ground Processing Scheduling Program.
- Developed at Kennedy Space Center
- Preparing a Shuttle for a new flight – a dynamic scheduling problem.
- GPSS schedules Space Shuttle maintenance based on available personnel, time, and resources.
- Uses a technique called constraint based iterative repair.

# Search (cont)

**Best-First Search**

1. Set **N** to be a list of initial nodes.
2. If **N** is empty, then exit and signal failure.
3. Set **n** to be the first node in **N**, and remove **n** from **N**.
4. If **n** is a goal node, then exit and signal success.
5. Otherwise, add the children of **n** to **N**, sort the nodes in **N** according to their estimated distance from a goal, and return to step 2.

# BinaryPredicate.java

```
public interface BinaryPredicate {

public boolean
 evaluate(SearchNode object1, SearchNode object2);

}
```

# UniformCostPredicate.java

```
/** Comparison function for
uniform-cost search nodes.
*/
public final class
   UniformCostPredicate
      implements BinaryPredicate {
 public boolean
  evaluate(SearchNode object1, SearchNode object2)
    return object1.getPathCost()
       > object2.getPathCost();
    }
}
```

# PriorityQueue.java

```
import java.lang.*;
import java.util.*;
public class PriorityQueue
    implements GeneralQueue {
   protected ArrayList<SearchNode> queue;
   protected BinaryPredicate comparator;
   public PriorityQueue
       (BinaryPredicate comparator) {
       queue = new ArrayList<SearchNode>();
       this.comparator = comparator;
   }
```

# PriorityQueue.java (cont)

```
public void add(SearchNode object) {
  int i;
  for (i=queue.size()-1; i>=0; i--) {
     if (comparator.evaluate
           (queue.get(i), object))
           break;}
   queue.add(object, i+1);}
public SearchNode removeFront() {
   int final = queue.size() -1;
   SearchNode object= queue.get(final);
   queue.remove(final);
   return object;}
public boolean isEmpty() {
  return queue.size() == 0;}}
```

# UniformCostSearch.java

```
import java.lang.*;
import java.util.*;
/**
    Uniform-cost search: redefines
     constructor to use priority queue
    with uniform-cost predicate.
*/
public class UniformCostSearch
      extends GeneralQueueSearch {
  public UniformCostSearch(State startState) {
   super(new SearchNode(startState),
new PriorityQueue(new UniformCostPredicate()));
    }
}
```

# HeuristicSearchNode.java

```
import java.lang.*;
import java.util.*;
/** This class extends SearchNode
   to include initial computation,
   storage, and retrieval of
   heuristic information.*/
public class HeuristicSearchNode
               extends SearchNode {
/** Value of heuristic for search node;*/
   protected float h;
/**No-argument constructor needed
      for Class.newInstance();*/
   public HeuristicSearchNode() { }
```

# HeuristicSearchNode.java (cont)

```
/**Constructor makes search node for
startState and computes and stores heuristic.*/
   public HeuristicSearchNode
            (State startState) {
      super(startState);
      computeH();n
   }
/**Computes and stores heuristic
           function for state. */
   protected void computeH() {
      h = ((Heuristic)state).h();
   }
/** Returns value of heuristic function.*/
   public float getH() {
      return h;}
```

# HeuristicSearchNode.java (cont)

```
/**  Returns f(node), defined as
   heuristic + cost to node.*/
   public float getF() {
      return pathCost+h;}
/** Returns a new node based
   on a successor of this node.*/
   public SearchNode
     makeNode(Successor successor) {
        HeuristicSearchNode node
          = (HeuristicSearchNode)
               super.makeNode(successor);
        node.computeH();
        return node;}
}
```

# GreedyPredicate.java

```
/**
    Comparison function
    for greedy search nodes.
*/

public final class GreedyPredicate
          implements BinaryPredicate {
 public boolean
   evaluate(SearchNode object1, SearchNode object2)
 return
   ( (HeuristicSearchNode) object1).getH()
    >
   ( (HeuristicSearchNode) object2).getH();
    }
}
```

# GreedySearch.java

```
import java.lang.*;
import java.util.*;
/**Greedy search: redefines
    constructor to use priority queue
    with Greedy predicate.*/
public class GreedySearch
   extends GeneralQueueSearch {
 public GreedySearch(State startState) {
  super((SearchNode)
 (new HeuristicSearchNode(startState)),
  new PriorityQueue(new GreedyPredicate()));
    }
}
```

# AStarPredicate.java

```
/**
   Comparison function for A* search nodes.
*/
public final class AStarPredicate
        implements BinaryPredicate {

  public boolean
  evaluate(SearchNode object1, SearchNode object2)
   return
     ( (HeuristicSearchNode) object1).getF()
     >
     ( (HeuristicSearchNode) object2).getF();
    }
}
```

# AStarSearch.java

```
import java.lang.*;
import java.util.*;
/**
    A* search: redefines
    constructor to use priority queue
    with A* predicate.
*/
public class AStarSearch
         extends GeneralQueueSearch {
    public AStarSearch(State startState) {
        super(new HeuristicSearchNode(startState),
     new PriorityQueue(new AStarPredicate()));
    }
}
```

# Driver

```
listPath( ( new UniformCostSearch(new
      TwoThreeState()) ).search() );
System.out.println();
System.out.println("Iterated Deepening Search:");
System.out.println();
System.out.println("A* Search:");
listPath( ( new AStarSearch(new
         TwoThreeState()) ).search() );
System.out.println();
  System.out.println("Greedy Search:");
listPath( ( new GreedySearch(new
      TwoThreeState()) ).search() );
System.out.println();
```

# Output

```
State:(23) Depth: 11 Cost: 24.0 by applying add3
State:(20) Depth: 10 Cost: 20.0 by applying add2
State:(18) Depth: 9 Cost: 18.0 by applying add2
State:(16) Depth: 8 Cost: 16.0 by applying add2
State:(14) Depth: 7 Cost: 14.0 by applying add2
State:(12) Depth: 6 Cost: 12.0 by applying add2
State:(10) Depth: 5 Cost: 10.0 by applying add2
State:(8) Depth: 4 Cost: 8.0 by applying add2
State:(6) Depth: 3 Cost: 6.0 by applying add2
State:(4) Depth: 2 Cost: 4.0 by applying add2
State:(2) Depth: 1 Cost: 2.0 by applying add2
Starting at state: (0)
```

# Output (cont)

```
A* Search:
State:(23) Depth: 11 Cost: 24.0 by applying add3
State:(20) Depth: 10 Cost: 20.0 by applying add2
State:(18) Depth: 9 Cost: 18.0 by applying add2
State:(16) Depth: 8 Cost: 16.0 by applying add2
State:(14) Depth: 7 Cost: 14.0 by applying add2
State:(12) Depth: 6 Cost: 12.0 by applying add2
State:(10) Depth: 5 Cost: 10.0 by applying add2
State:(8) Depth: 4 Cost: 8.0 by applying add2
State:(6) Depth: 3 Cost: 6.0 by applying add2
State:(4) Depth: 2 Cost: 4.0 by applying add2
State:(2) Depth: 1 Cost: 2.0 by applying add2
Starting at state: (0)
```

# Output (cont)

```
Greedy Search:
State:(23) Depth: 8 Cost: 30.0 by applying add2
State:(21) Depth: 7 Cost: 28.0 by applying add3
State:(18) Depth: 6 Cost: 24.0 by applying add3
State:(15) Depth: 5 Cost: 20.0 by applying add3
State:(12) Depth: 4 Cost: 16.0 by applying add3
State:(9) Depth: 3 Cost: 12.0 by applying add3
State:(6) Depth: 2 Cost: 8.0 by applying add3
State:(3) Depth: 1 Cost: 4.0 by applying add3
Starting at state: (0)
```