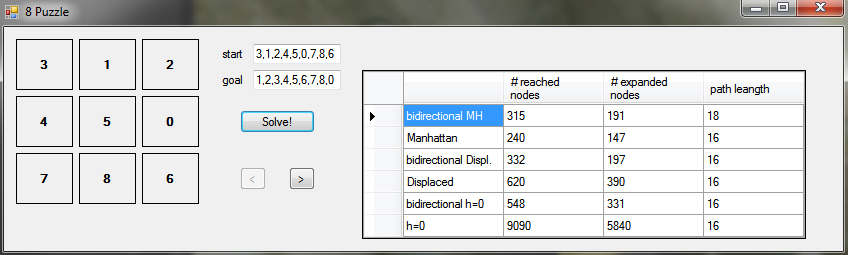
**A Star**

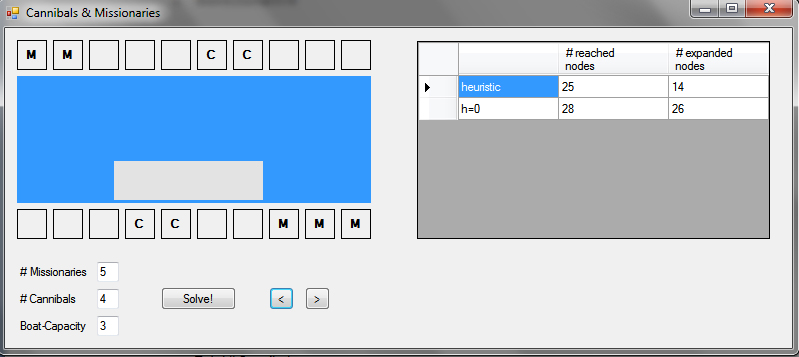
We wrote a reusable A Star code with parent-pointer redirection in C++ which could be used with minor changes in all the following applications using diverse state-definitions and heuristics. The states are implemented as vector<set> and a came\_from map is stored to enable back-tracing.

**8-Puzzle**



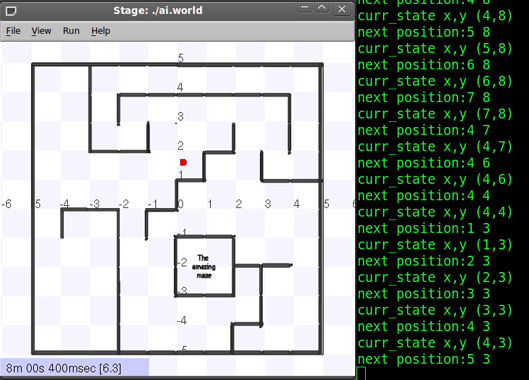
* Problem statement:
  + Implementation of the 8-puzzle problem which should be solved using several heuristics and the AStar algorithm.
  + The optimal path should be shown and in the GUI navigation through the optimal path should be possible
  + Heuristics should be shown with their performances and be presented in a comparable manner
* Inputs:
  + Start-State
  + Goal-State
* State definition:
  + 9 fields with integers (0-8)
* Heuristics used:
  + Manhattan
  + Bidirectional Manhattan
  + Displaced Tiles
  + Bidirectional Displaced Tiles
  + h=0
  + Bidirectional h=0
* Observations:
  + Manhattan heuristic works best most of the time (1-2 order(s) less nodes visited and expanded)
  + Bidirectional search does not always find the optimal path
  + Bidirectional search does not always terminate faster than original (e.g. with Manhattan heuristic)
  + Surprisingly bidirectional search with h=0 reduces the search significantly

**Missionaries and Cannibals**



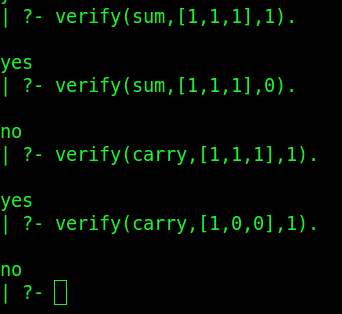
* Problem Statement:
  + The missionaries and Cannibals problem should be solved using the same AStar algorithm as in the 8-Puzzle assignment
  + Number of Cannibals, Missionaries and the boat capacity should be variable (here constrained to 5M and 5C, due to GUI)
  + User should be able to navigate in GUI through the optimal path to see the solution
* Inputs:
  + # Missionaries
  + # Cannibals
  + Boat-Capacity
* State definition:
  + # Missionaries on upper shore (x[0])
  + # Cannibals on upper shore (x[1])
  + Boat-position (up/down)
  + Total # Missionaries
  + Total # Cannibals
* Heuristics used:
  + h =(x[0] + x[1])/2) + 1
  + h=0
* Observations:
  + The code can handle any amount of cannibals/missionaries and different boat-capacities and is only limited in our program to be displaced nicely
  + Heuristic works well with increasing number of cannibals/missionaries/boat-capacity
  + The heuristic reaches a similar amount of nodes but expands only a fraction

**Player-Stage**

****

* Problem Statement:
  + A robot should be able to navigate through a maze using the AStar algorithm.
  + After finding the goal it should trace back the optimal path to its initial start point.
* Inputs:
  + Any 10x10 solvable maze in which the starting point is (1x1) and the goal (10x10)
* Robot design:
  + Robot with 4 sonar-sensors (front, left, right, rear)
  + Absolute x,y position and angle is known, as well as the absolute goal position
  + Movement realized with a P-Controller
  + Robot turns into the right direction and then moves to reach the next square (to its north/east/west/south)
* State definition:
  + x-Position (in squares) on grid
  + y-Position (in squares) on grid
* Heuristic used:
  + Manhattan
  + Uses min(h-value) as a secondary criterion when several states have the same f-value
* Observations:
  + A Star finds the optimal path from the start to the goal
  + The velocity of this method to find the goal fast depends on the nature of the maze: Sometimes the smallest f-value is far away from the robot, so that it has to move long distances to keep exploring. On the other hand this method ensures to have found the optimal path in the end (which then can for example just be traced back to return to the starting point). Also by storing and making use of information we can assure that the goal will be found (if theoretically possible).
  + This method would be difficult to implement in a realistic robot, as the absolute position and angle are usually not known and with relative maneuvering errors keep adding up.

**Prolog circuit verification**

**

* Problem Statement:
  + Using Prolog, check whether a circuit works or not.
  + Input consists of gates and variables present in the circuit (input list and output).
  + Implement the logic for AND, OR, NAND & NOT gates.
  + Implement more complex circuits using above gates

*general.pl*consists of the following relations

* The basic logic gates *AND,OR,NOT*
* Recursive relation to verify any circuit. It has the format   
   *verify(<gate-to-be verified>,<input-list>,<output>)*
* Relation *elementAt(<index>,<list>,<output>). Output*has the element at index position in the list. List is assumed to be 0 indexed.

The relations of the gates to be verified are in specific files *‘my\_xor.pl’* and *‘fulladder.pl’*. ´These files contain the relations of how the circuits have been defined.

The relations are

* gate(<name-of-gate>,x): relates the gate to x.
* terminal(gate,1/2/3,y): relates the gate ‘gate’ terminals to the connection line, i.e. the terminal 1/2/3 is connected to which of the wires carrying the signal.   
  For 2 input gate, terminal 1,2 are input terminals and 3 is the output terminal.  
  For 1 input gate, terminal 1 is the input terminal and terminal 2 is the output terminal.

RESULTS AND TEST CASES:

include the file containing the description of the circuit. We have implemented 'fulladder.pl' and 'palindrome\_3bit.pl'. This is done by uncommenting the includes in the 'general.pl'. Input will be of the form

verify(palindrome,[1,1,1],1).

will give a result yes

verify(palindrome,[1,0,1],1).

will give a result yes.

verify(palindrome,[1,0,0],1).

will give a result no.

verify(parity,[1,1,1],1)

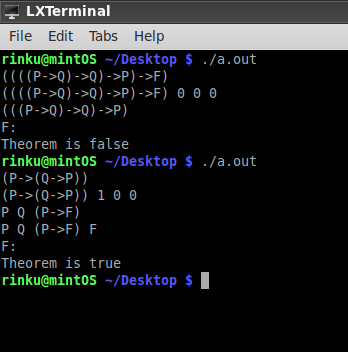
will give a result yes.

verify(parity,[1,1,0],1)

will give a result no.

NOTE: include only one file at a time to check the circuits.

**Theorem prover**



* Written in c++
* Pseudocode:

*Read\_Input();*

*Format\_Input(); // checks for invalid char and remove ‘->’*

*Create\_recursively\_tree(); // nodes are being created depending on brackets*

*Apply\_deduction\_theorem\_on\_tree(); // recursively testable statements are created*

*Apply\_modus\_ponens(); // MP used for all statements*

*Verify{*

*if provable\_through\_Axioms() return true; //true statements can be derived using A1/A2/A3*

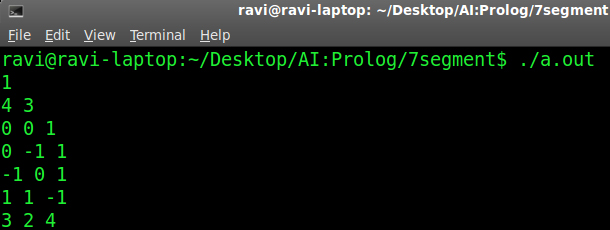
*if Compare\_Statements() return true; // compares existing statements to A1,A2,A3*

*if derived\_statements() return true; // checks if derived statements are true*

*return false;*

*}*

**Perceptron Training**



* Implementation:
  + Generalized implementation for a feed-forward neural network trained with back propagation
  + Learning rate = 0.6
  + Random initial weight initialization [0,5]
  + Error tolerance = 0.1
  + Palindrome, Parity can be modeled
* Inputs:
  + # Hidden layers
  + # inputs
  + Truthtable
* Outputs:
  + New weights