## AI Through Search

1. The number of cells visited was much lower with the Bidirectional search than with the regular UCS search.

static bool UCS() {

ArrayList path;

if (UCSForward()) {

path = solPath;

if (UCSBackwards()) {

path.AddRange(solPath);

return true;

}

}

return false;

}

static bool UCSForward() {

PriorityQueue frontier = new PriorityQueue();

explored = new HashSet<Position>();

paths = new ArrayList();

// Add initial State to frontier

foreach (int x in (ArrayList)maze[0])

if (x > 0) {

int pos = ((ArrayList)maze[0]).IndexOf(x);

frontier.Enqueue(new Position(0, pos, (int)((ArrayList)maze[0])[pos]));

break;

}

// Find Goal

Position goal = new Position(0, 0);

foreach (int x in (ArrayList)maze[maze.Count - 1])

if (x > 0) {

int pos = ((ArrayList)maze[maze.Count - 1]).IndexOf(x);

goal = new Position(maze.Count - 1, pos, (int)((ArrayList)maze[maze.Count - 1])[pos]);

break;

}

bool found = false;

Position curPos = new Position(0, 0);

while (curPos.y < maze.Count/2) {

// explore a new position

curPos = frontier.Dequeue();

// Have we found the goal?

if (curPos.y >= maze.Count / 2) {

found = true;

break;

}

explored.Add(curPos);

addToPath(curPos);

// Find new states for the frontier

ArrayList tempFront = new ArrayList();

if (curPos.y - 1 < ((ArrayList)maze[0]).Count)

tempFront.Add(new Position(curPos.x, curPos.y - 1, curPos.cost + (int)((ArrayList)maze[curPos.x])[curPos.y - 1]));

if (curPos.x + 1 < maze.Count)

tempFront.Add(new Position(curPos.x + 1, curPos.y, curPos.cost + (int)((ArrayList)maze[curPos.x + 1])[curPos.y]));

if (curPos.y + 1 >= 0)

tempFront.Add(new Position(curPos.x, curPos.y + 1, curPos.cost + (int)((ArrayList)maze[curPos.x])[curPos.y + 1]));

if (curPos.x - 1 >= 0)

tempFront.Add(new Position(curPos.x - 1, curPos.y, curPos.cost + (int)((ArrayList)maze[curPos.x - 1])[curPos.y]));

// Add new states to the frontier

foreach (Position pos in tempFront)

if ((int)((ArrayList)maze[pos.x])[pos.y] > 0 && !hashContains(pos, explored)) {

if (!priorityQueueContains(pos, frontier))

frontier.Enqueue(pos);

else {

foreach (Position pos2 in frontier)

if (pos.x == pos2.x && pos.y == pos2.y)

if (pos.cost < pos2.cost)

pos2.cost = pos.cost;

}

}

if (frontier.Count > memory)

memory = frontier.Count;

}

// Found the goal

if (found) {

// Find the final path

solPath = (ArrayList)paths[paths.Count - 1];

return true;

}

// Didn't find the goal

return false;

}

static bool UCSBackwards() {

PriorityQueue frontier = new PriorityQueue();

explored = new HashSet<Position>();

paths = new ArrayList();

// Add initial State to frontier

foreach (int x in (ArrayList)maze[maze.Count - 1])

if (x > 0) {

int pos = ((ArrayList)maze[maze.Count - 1]).IndexOf(x);

frontier.Enqueue(new Position(maze.Count - 1, pos, (int)((ArrayList)maze[maze.Count - 1])[pos]));

break;

}

// Find Goal

Position goal = new Position(0, 0);

foreach (int x in (ArrayList)maze[0])

if (x > 0) {

int pos = ((ArrayList)maze[0]).IndexOf(x);

goal = new Position(0, pos, (int)((ArrayList)maze[0])[pos]);

break;

}

bool found = false;

Position curPos = new Position(0, 0);

while (curPos.y >= maze.Count / 2) {

// explore a new position

curPos = frontier.Dequeue();

// Have we found the goal?

if (curPos.y < maze.Count / 2) {

found = true;

break;

}

explored.Add(curPos);

addToPath(curPos);

// Find new states for the frontier

ArrayList tempFront = new ArrayList();

if (curPos.y - 1 < ((ArrayList)maze[0]).Count)

tempFront.Add(new Position(curPos.x, curPos.y - 1, curPos.cost + (int)((ArrayList)maze[curPos.x])[curPos.y - 1]));

if (curPos.x + 1 < maze.Count)

tempFront.Add(new Position(curPos.x + 1, curPos.y, curPos.cost + (int)((ArrayList)maze[curPos.x + 1])[curPos.y]));

if (curPos.y + 1 >= 0)

tempFront.Add(new Position(curPos.x, curPos.y + 1, curPos.cost + (int)((ArrayList)maze[curPos.x])[curPos.y + 1]));

if (curPos.x - 1 >= 0)

tempFront.Add(new Position(curPos.x - 1, curPos.y, curPos.cost + (int)((ArrayList)maze[curPos.x - 1])[curPos.y]));

// Add new states to the frontier

foreach (Position pos in tempFront)

if ((int)((ArrayList)maze[pos.x])[pos.y] > 0 && !hashContains(pos, explored)) {

if (!priorityQueueContains(pos, frontier))

frontier.Enqueue(pos);

else {

foreach (Position pos2 in frontier)

if (pos.x == pos2.x && pos.y == pos2.y)

if (pos.cost < pos2.cost)

pos2.cost = pos.cost;

}

}

if (frontier.Count > memory)

memory = frontier.Count;

}

// Found the goal

if (found) {

// Find the final path

solPath = (ArrayList)paths[paths.Count - 1];

solPath.Reverse();

return true;

}

// Didn't find the goal

return false;

}

static void addToPath(Position newPos) {

bool placed = false;

if (paths.Count == 0) {

ArrayList newList = new ArrayList();

newList.Add(newPos);

paths.Add(newList);

return;

}

// Check the paths and add

int i = 0;

while (!placed) {

ArrayList list = (ArrayList)paths[i];

if (canReach(newPos, (Position)list[list.Count - 1])) {

paths.Add((ArrayList)list.Clone());

list.Add(newPos);

placed = true;

}

i++;

}

}

## Formal Logic

1. A1: p => q Premise 1

A2: ¬p => ¬q ^ r Premise 2

A3: q V r => s Premise 3

A4: s => t Premise 4

A5: ¬s Negation of the consequence

A6: ¬s => ¬(q V r) Conditional rule 2 using A3

A7: ¬(q V r) Detachment using A5 and A6  
A8: ¬q ^ ¬r De Morgan’s Law using A7

A9: ¬q ^-elimination using A8

A10: ¬q => ¬p Conditional rule 2 using A1

A11: ¬q => ¬q ^ r Chain rule using A10 and A2

A12: ¬q ^ r Detachment using A11 and A9

A13: ¬r Commutative law and ^-elimination using A8

A14: r Commutative law and ^-elimination using A12

1. Step1: Convert to Standard Form

¬p, p V r, ¬q V ¬r, q V s |= (p ∧ q) V (¬p ∧ ¬q)

Step2: Negation Removal

p V r, ¬q V ¬r, q V s |= (p ∧ q) V (¬p ∧ ¬q), p

Step3: And/Or Removal

p V r, ¬q V ¬r, q V s |= p ∧ q, ¬p ∧ ¬q, p

Step4: Theorem Splitting

1. p, ¬q V ¬r, q V s |= p ∧ q, ¬p ∧ ¬q, p

True because p exists on both side

1. r, ¬q V ¬r, q V s |= p ∧ q, ¬p ∧ ¬q, p

Step1: Theorem Splitting

i) r, ¬q, q V s |= p ∧ q, ¬p ∧ ¬q, p

Step 1: Negation Removal

r, q V s |= p ∧ q, ¬p ∧ ¬q, p, r, q

Step 2: Theorem Splitting

r, q V s |= p ∧ q, ¬p, p, r, q and r, q V s |= p ∧ q, ¬q, p, r, q

Step 3: Negation Removal

r, q V s, p |= p ∧ q, p, r, q and r, q V s, q |= p ∧ q, p, r, q

Both are true

ii) r, ¬r, q V s |= p ∧ q, ¬p ∧ ¬q, p

Step 1: Negation Removal

r, q V s |= p ∧ q, ¬p ∧ ¬q, p, r

True since r appears on both sides

∎

1. [¬P(x, y) V ¬R(y)] V [Q(f(x), h(x)) ∧ R(g(x), j(x))]
2. C1: M(x) ∨ Q(x)

C2: ¬M(Clive)

C3: ¬Q(y)

C4: Q(Clive) (C1 and C2 with sub {Clive|x})

C5: ∎ (C3 and C4 with sub {Clive|y})

## Prolog

1. 1. Rule: brother(X, Y) :- parent(Z, X), parent(Z, Y).

Query: brother(ushaka, X).

The brothers of Ushaka are Mpande and Dingane.

* 1. Rule: uncle(X, Y) :- brother(Z, X), parent(Z, Y).

Query: uncle(\_, X).

The following had uncles who were rulers:

dinuzulu

mswatiIII

thumbumuzi

mcwayizeni

nyangayezizwe

cetshwayo

sobhuza

goodwill

ngwane

solomon

athur

* 1. Rule: related(X, Y) :- brother(X, Y); uncle(X, Y); parent(X, Y); parent(Y, X).  
     Query: related(ushaka, goodwill).  
     Ushaka is not related to Goodwill

Query: related(senzangakhona, dingane).

Senzangakhona is related to Dingane

Query: related(dinuzulu, mbandzeni).

Dinuzulu is not related to Mbandzeni