25. Nov. 2004

Business Intelligence 2004/05

Sample Solutions for Assignment #1: Search, etc.

Part 2: Solving a problem with CSP

a)

House #1	House #2	House #3	House #4	House #5
Yellow	Blue	White	Green	Red
Norwegian	Chinese	Russian	Portuguese	English
Water	Wilk	Tea	Coffee	OJ
Marlboro	Parisienne	Barclay	Chesterfield	Luckies
Zebra	Horse	Rabbit	Dog	Fox

b)

Variables:

- Yellow, Red, Blue, White, Green
- Zebra, Dog, Fox, Rabbit, Horse
- Marlboro, Chesterfield, Parisienne, Lucky Strike, Barclay
- Englishman, Portuguese, Norwegian, Russian, Chinese
- Orange Juice, Water, Tea, Coffee, Milk

Domains of all Variables: {1,2,3,4,5}

Constraints:

- English = Red
- Portuguese = Dog
- Norwegian = 1
- Marlboro = Yellow
- Chesterfield = Fox + 1 OR Fox -1
- Norwegian = Blue + 1 OR Blue – 1
- Barclay = Rabbit
- Luckies = OJ
- Russian = Tea
- Chinese = Parisienne
- Marlboro = Horse + 1 OR Horse -1

Jiwen Li (li@ifi.unizh.ch), Christoph Kiefer (kiefer@ifi.unizh.ch)

- Coffee = Green
- Green = White + 1
- Milk = Yellow + 1

c)

1. The Norwegian lives in the first house on the left. => Split the domain of the Norwegian to only contain the value {1}.
2. It its clear from constraint number 6 that the second house from the left must be the blue house. => Split the domain of Blue to only contain the value {2}.
3. Run the applet.
4. The variable White now contains the values {3,4}. Make a guess and split it's domain to only contain value {3} for instance. => The applet finds the correct solution (compare figures 1 and 2).

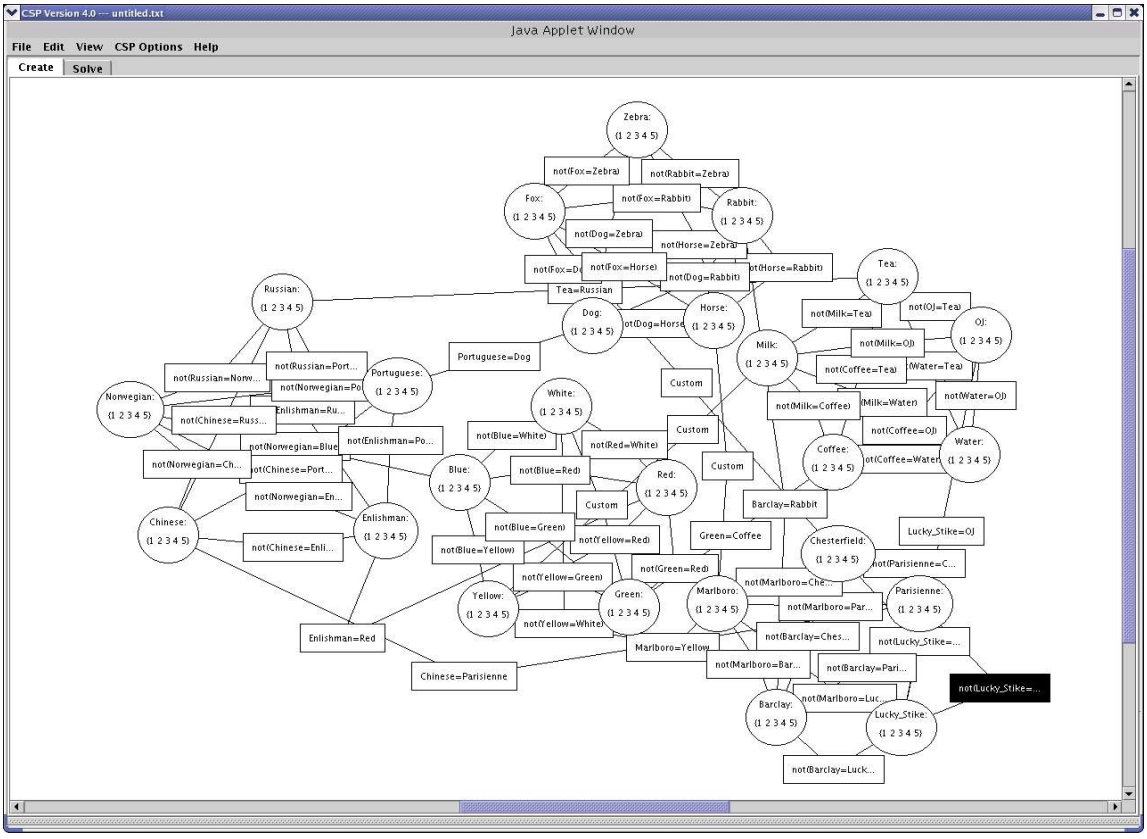

Figure 1

Jiwen Li (li@ifi.unizh.ch), Christoph Kiefer (kiefer@ifi.unizh.ch)

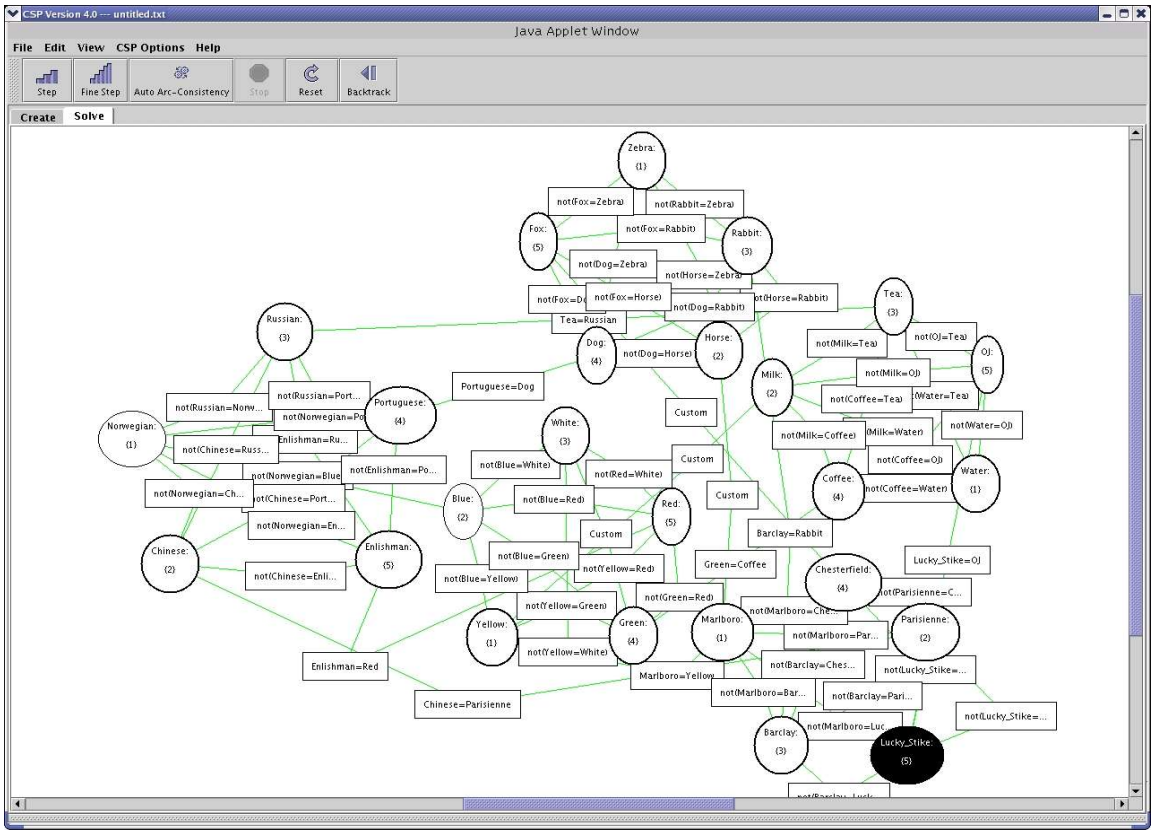

Figure 2

Jiwen Li (li@ifi.unizh.ch), Christoph Kiefer (kiefer@ifi.unizh.ch)

25. Nov. 2004

Part 3: Programming a search problem

a)

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

We encode the frame checker locations as shown in the graph above. For example, 0 represents the top-left checker in the frame. We use capital letters B followed by a number ε [0,10] to represent each block in the puzzle frame. For example, B0 represents block number 0 in the above frame. We use animal names to represent the shapes of different blocks.

Block Puzzle 1:

Init(At(B0,1) ^ At(B1,0) ^ At(B2,4) ^ At(B3,3) ^ At(B4,7) ^ At(B5,8) ^ At(B6,10) ^ At(B7,12) ^ At(B8,14) ^ At(B9,16) ^ At(B10,19) ^ Is(B0,SPIDER) ^ Is(B1,LADYBUG) ^ Is(B2,LADYBUG) ^ Is(B3,LADYBUG) ^ Is(B4,LADYBUG) ^ Is(B5,FISH) ^ Is(B6,FISH) ^ Is(B7,FISH) ^ Is(B8,FISH) ^ Is(B9,LADYBUG) ^ Is(B10,LADYBUG) ^ Clear(17) ^ Clear(18) ^ Checker(0) ^ Checker(1) ^ ... ^ Checker(19) ^ Block(B0) ^ Block(B1) ^ ... ^ Block(B10) ^ Shape(LADYBUG) ^ Shape(SPIDER) ^ Shape(FISH))

Goal(At(B0,13))

Action(Move_left_LADYBUG(b,c)
PRECOND: Is(b,LADYBUG) ^ Shape(LADYBUG) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker(c -1) ^ Clear(c -1)
EFFECT: Clear(c) ^ At(b,c -1))

Action(Move_right_LADYBUG(b,c)
PRECOND: Is(b,LADYBUG) ^ Shape(LADYBUG) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker(c +1) ^ Clear(c +1)
EFFECT: Clear(c) ^ At(b,c + 1))

Jiwen Li (li@ifi.unizh.ch), Christoph Kiefer (kiefer@ifi.unizh.ch)

Action(Move_up_LADYBUG(b,c)
PRECOND: Is(b, LADYBUG) ^ Shape(LADYBUG) ^ At(b,c) ^ Block(b) ^ Checker (c) ^ Checker(c - 4) ^ Clear(c - 4)
EFFECT: Clear(c) ^ At(b,c - 4))

Action(Move_down_LADYBUG(b,c)
PRECOND: Is(b,LADYBUG) ^ Shape(LADYBUG) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker(c + 4) ^ Clear(c + 4)
EFFECT: Clear(c) ^ At(b,c + 4))

Action(Move_left_SPIDER(b,c)
PRECOND: Is(b,SPIDER) ^ Shape(SPIDER) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker(c - 1) ^ Clear(c - 1) ^ Checker(c + 3) ^ Clear(c + 3)
EFFECT: Clear(c + 1) ^ Clear(c + 5) ^ At(b,c -1))

Action(Move_right_SPIDER(b,c)
PRECOND: Is(b,SPIDER) ^ Shape(SPIDER) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker(c + 2) ^ Clear(c + 2) ^ Checker(c + 5) ^ Clear(c + 5)
EFFECT: Clear(c) ^ Clear(c + 4) ^ At(b,c + 1))

Action(Move_up_SPIDER(b,c)
PRECOND: Is(b,SPIDER) ^ Shape(SPIDER) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker(c - 4) ^ Clear(c - 4) ^ Checker(c - 3) ^ Clear(c - 3)
EFFECT: Clear(c + 4) ^ Clear(c + 5) ^At(b,c - 4))

Action(Move_down_SPIDER(b,c)
PRECOND: Is(b,SPIDER) ^ Shape(SPIDER) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker(c + 8) ^ Clear(c + 8) ^ Checker(c + 9) ^ Clear(c + 9)
EFFECT: Clear(c) ^ Clear(c + 1) ^At(b,c + 4))

Action(Move_left_FISH(b,c)
PRECOND: Is(b,FISH) ^ Shape(FISH) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker (c - 1) ^ Clear(c - 1)
EFFECT: Clear(c + 1) ^ At(b,c -1))

Action(Move_right_FISH(b,c)
PRECOND: Is(b,SPIDER) ^ Shape(SPIDER) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker(c + 2) ^ Clear(c + 2)
EFFECT: Clear(c) ^ At(b,c + 1))

Action(Move_up_FISH(b,c)
PRECOND: Is(b,FISH) ^ Shape(FISH) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker (c - 4) ^ Clear(c - 4) ^ Checker(c - 3) ^ Clear(c - 3)
EFFECT: Clear(c) ^ Clear(c + 1) ^At(b,c - 4))

Jiwen Li (li@ifi.unizh.ch), Christoph Kiefer (kiefer@ifi.unizh.ch)

Action(Move_down_FISH(b,c)
PRECOND: Is(b,FISH) ^ Shape(FISH) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker (c + 4) ^ Clear(c + 4) ^ Checker(c + 5) ^ Clear(c + 5)
EFFECT: Clear(c) ^ Clear(c + 1) ^At(b,c + 4))

Block Puzzle 2:

Init(At(B0,0) ^ At(B1,2) ^ At(B2,3) ^ At(B3,6) ^ At(B4,10) ^ At(B5,11) ^ At(B6,14) ^ At(B7,18) ^ At(B8,19) ^ At(B9,12) ^ Is(B0,SPIDER) ^ Is(B1,LADYBUG) ^ Is (B2,LADYBUG) ^ Is(B3,FISH) ^ Is(B4,LADYBUG) ^ Is(B5,LADYBUG) ^ Is (B6,FISH) ^ Is (B7,LADYBUG) ^ Is(B8,LADYBUG) ^ Is(B9,WORM) ^ Clear(8) ^ Clear(9) ^ Checker (0) ^ Checker(1) ^ ... ^ Checker(19) ^ Block(B0) ^ Block(B1) ^ ... ^ Block(B9) ^ Shape (LADYBUG) ^ Shape(SPIDER) ^ Shape(FISH) ^ Shape (WORM))

Goal(At(B0,12) ^ At(B9,0))

Action(Move_left_LADYBUG(b,c)
PRECOND: Is(b,LADYBUG) ^ Shape(LADYBUG) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker(c - 1) ^ Clear(c - 1)
EFFECT: Clear(c) ^ At(b,c -1))

Action(Move_right_LADYBUG(b,c)
PRECOND: Is(b,LADYBUG) ^ Shape(LADYBUG) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker(c + 1) ^ Clear(c + 1)
EFFECT: Clear(c) ^ At(b,c + 1))

Action(Move_up_LADYBUG(b,c)
PRECOND: Is(b,LADYBUG) ^ Shape(LADYBUG) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker(c - 4) ^ Clear(c - 4)
EFFECT: Clear(c) ^ At(b,c - 4))

Action(Move_down_LADYBUG(b,c)
PRECOND: Is(b,LADYBUG) ^ Shape(LADYBUG) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker(c + 4) ^ Clear(c + 4)
EFFECT: Clear(c) ^ At(b,c + 4))

Action(Move_left_SPIDER(b,c)
PRECOND: Is(b,SPIDER) ^ Shape(SPIDER) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker(c - 1) ^ Clear(c - 1) ^ Checker(c + 3) ^ Clear(c + 3)
EFFECT: Clear(c + 1) ^ Clear(c + 5) ^ At(b,c -1))

Jiwen Li (li@ifi.unizh.ch), Christoph Kiefer (kiefer@ifi.unizh.ch)

Action(Move_right_SPIDER(b,c)
PRECOND: Is(b,SPIDER) ^ Shape(SPIDER) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker(c + 2) ^ Clear(c + 2) ^ Checker(c + 5) ^ Clear(c + 5)
EFFECT: Clear(c) ^ Clear(c + 4) ^ At(b,c + 1))

Action(Move_up_SPIDER(b,c)
PRECOND: Is(b,SPIDER) ^ Shape(SPIDER) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker(c - 4) ^ Clear(c - 4) ^ Checker(c - 3) ^ Clear(c - 3)
EFFECT: Clear(c + 4) ^ Clear(c + 5) ^At(b,c - 4))

Action(Move_down_SPIDER(b,c)
PRECOND: Is(b,SPIDER) ^ Shape(SPIDER) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker(c + 8) ^ Clear(c + 8) ^ Checker(c + 9) ^ Clear(c + 9)
EFFECT: Clear(c) ^ Clear(c + 1) ^At(b,c + 4))

Action(Move_left_WORM(b,c)
PRECOND: Is(b,WORM) ^ Shape(WORM) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker(c - 1) ^ Clear(c - 1) ^ Checker(c + 3) ^ Clear(c + 3)
EFFECT: Clear(c + 1) ^ Clear(c + 5) ^ At(b,c -1))

Action(Move_right_WORM(b,c)
PRECOND: Is(b,WORM) ^ Shape(WORM) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker(c + 2) ^ Clear(c + 2) ^ Checker(c + 5) ^ Clear(c + 5)
EFFECT: Clear(c) ^ Clear(c + 4) ^ At(b,c + 1))

Action(Move_up_WORM(b,c)
PRECOND: Is(b,WORM) ^ Shape(WORM) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker(c - 4) ^ Clear(c - 4) ^ Checker(c - 3) ^ Clear(c - 3)
EFFECT: Clear(c + 4) ^ Clear(c + 5) ^At(b,c - 4))

Action(Move_down_WORM(b,c)
PRECOND: Is(b,WORM) ^ Shape(WORM) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker(c + 8) ^ Clear(c + 8) ^ Checker(c + 9) ^ Clear(c + 9)
EFFECT: Clear(c) ^ Clear(c + 1) ^At(b,c + 4))

Action(Move_left_FISH(b,c)
PRECOND: Is(b,FISH) ^ Shape(FISH) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker (c - 1) ^ Clear(c - 1)
EFFECT: Clear(c + 1) ^ At(b,c -1))

Action(Move_right_FISH(b,c)
PRECOND: Is(b,SPIDER) ^ Shape(SPIDER) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker(c + 2) ^ Clear(c + 2)
EFFECT: Clear(c) ^ At(b,c + 1))

Jiwen Li (li@ifi.unizh.ch), Christoph Kiefer (kiefer@ifi.unizh.ch)

Action(Move_up_FISH(b,c)
PRECOND: Is(b,FISH) ^ Shape(FISH) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker (c - 4) ^ Clear(c - 4) ^ Checker(c - 3) ^ Clear(c - 3)
EFFECT: Clear(c) ^ Clear(c + 1) ^At(b,c - 4))

Action(Move_down_FISH(b,c)
PRECOND: Is(b,FISH) ^ Shape(FISH) ^ At(b,c) ^ Block(b) ^ Checker(c) ^ Checker (c + 4) ^ Clear(c + 4) ^ Checker(c + 5) ^ Clear(c + 5)
EFFECT: Clear(c) ^ Clear(c + 1) ^At(b,c + 4))

Jiwen Li (li@ifi.unizh.ch), Christoph Kiefer (kiefer@ifi.unizh.ch)

b)

1. Breadth-First Search

```
function BREATH-SEARCH(initial_status, goal_status, actions) returns a solution, or failure
        return BFS(initial_status, goal_status, actions)

function BFS(initial_status, goal_status, actions) returns a solution or failure

        current_status = initial_status;
        add(current_status, status_queue);

        while status_queue != empty do

                for each action do
                        next_status = action(current_status);
                        if not visited(next_status) then
                                add(next_status, status_queue);
                        end if
                end for

                current_status = remove(status_queue);
                if current_status = goal_status then
                        add(current_status, solution)
                        break;
                end if

        end while

        if status_queue == empty then
                return failure;
        end if

        while hasparent(current_status) do
                current_status = current_status.parent;
                add(current_status, solution)
        end while

        return solution
```

Jiwen Li (li@ifi.unizh.ch), Christoph Kiefer (kiefer@ifi.unizh.ch)

2. Depth-First Search

```
function DEPTH-SEARCH(initial_status, goal_status, actions) returns a solution,
or failure
        return RECURSIVE-DFS(initial_status, goal_status, actions)

function RECURSIVE-DFS(initial_status, goal_status, actions) returns a solution
or failure

        current_status = initial_status;
        if current_status = goal_status then
                solution = current_status;
                return solution;
        else
                for each action do
                        next_status = action(current_status);
                        if not visited(next_status) then
                                result = RECURSIVE-DFS(next_status,
                                                       goal_status,
                                                       actions);
                                if result = failure then
                                        return failure;
                                else
                                        solution = current_status + result;
                                        return solution;
                                end if

                        end if
        end if
```

Jiwen Li (li@ifi.unizh.ch), Christoph Kiefer (kiefer@ifi.unizh.ch)

3. Iterative Deepening Depth-First Search

```
function ITERATIVE-DEEPENING-SEARCH(input) returns a solution or failure
inputs: initial_status, goal_status, actions;

        for depth from 0 to ∞ do
                result = DLS(initial_status, goal_status, actions, depth);
                if result != cutoff then
                        solution = result;
                        return solution.
                end if
        end for

function DLS(initial_status, goal_status, actions, limit) returns a solution or failure
/ cutoff
        return RECURSIVE-DLS(initial_status, goal_status, actions, limit)
```

Jiwen Li (li@ifi.unizh.ch), Christoph Kiefer (kiefer@ifi.unizh.ch)

function RECURSIVE-DLS(initial_status, goal_status, actions, limit) **returns** a solution or failure / cutoff

```
cutoff_occurred = false;
current_status = initial_status;
if current_status = goal_status then
        solution = current_status
        return solution;
else if depth(current_status) >= limit then
        return cutoff;
else
        for each action do
                next_status = action(current_status);
                if not visited(next_status) then
                        result = RECURSIVE-DLS(next_status,
                                                goal_status,
                                                actions)

                        if result = cutoff then
                                cutoff_occurred = true;
                                return cutoff;
                        else if result = failure then
                                return failure;
                        else
                                solution = current_status + result;
                                return solution
                        end if

                end if
        end for
end if
```

Jiwen Li (li@ifi.unizh.ch), Christoph Kiefer (kiefer@ifi.unizh.ch)

4. Greedy Best-First Search

Here, we define the **heuristic function** h(current_status) as the distance from the current_status to the goal_status.

We give the Greedy Best-First algorithm in pseudocode as follows:

```
function GREEDY-SEARCH(initial_status, goal_status, actions) returns a solution or failure
        return BFS(initial_status, goal_status, actions)

function BFS(initial_status, goal_status, actions) returns a solution, or failure

        current_status = initial_status;
        while current_status != goal_status and status_queue != empty do

                for each action do
                        next_status = action(current_status);
                        if not visited(next_status) then
                                new_status.fvalue = h(new_status);
                                add(next_status, status_queue);
                                sort(status_queue, fvalue);
                        end if
                end for

                current_status = remove(status_queue);

        end while

        if status_queue == empty then
                return failure;
        end if

        while hasparent(current_status) do
                current_status = current_status.parent;
                add(current_status, solution)
        end while

        return solution
```

Jiwen Li (li@ifi.unizh.ch), Christoph Kiefer (kiefer@ifi.unizh.ch)

5. A Search*

Here we define h(current_status) as the distance from the current_status to the goal_status. g(current_status) is defined as the minimum traveling steps from the initial_status to the current_status.

We give the A* Breath-First algorithm in pseudocode as follows:

function A*-SEARCH(initial_status, goal_status, actions) **returns** a solution, or failure

```
    return ABF(initial_status, goal_status, actions)
```

function ABF(initial_status, goal_status, actions) returns a solution, or failure

```
    current_status = initial_status;
    while current_status != goal_status and status_queue != empty do

        for each action do
            next_status = action(current_status);
            if not visited(next_status) then
                new_status.fvalue = h(new_status) + g(new_status);
                add(next_status, status_queue);
                sort(status_queue, fvalue);
            end if
        end for

    end while

    current_status = remove(status_queue);

    if status_queue == empty then
        return failure;
    end if

    while hasparent(current_status) do
        current_status = current_status.parent;
        add(current_status, solution)
    end while

    return solution
```

Jiwen Li (li@ifi.unizh.ch), Christoph Kiefer (kiefer@ifi.unizh.ch)

d)

1. Breadth-First Search

Completeness:
If the block puzzle problem has solutions (status searching space is limited), Breadth-First Search definitely finds an optimal one.

Space:
The total number of frame status' in memory for Breadth-First Search are approximately as b + b^2 + b^3 +…+ b^d + (b^(d+1)-b) = O(b^(d + 1)), where b is the number of actions (branching factor) and d is the number of steps the target block has moved (depth).

Time:
The total time for searching the status is also about O(b^(d+1)), so it is an exponential-complexity search problem. However, the memory requirements are a bigger problem for Breadth-First Search than is execution time.

Optimality:
Breadth-First Search is a brute-force searching strategy which only uses the predefined information of initial status, goal status and actions. There is no use of any heuristic function as evaluation function to guide the search. However, in the case of the block puzzle problem, the shallowest solution is the optimal solution, so this strategy finds at least one of the optimal solutions in this problem.

2. Depth-First Search

Completeness:
If the block puzzle problem has solution (status searching space is limited), Depth-First search definitely finds a one.

Space:
Depth-First Search has very modest memory requirements. For a block puzzle with m possible actions and a minimum of n steps to find the solution, the total number of frame status in memory is about m*n + 1, so it is a linear-complexity memory cost searching problem.

Time:
In the worst case, Depth-First Search will search all m^n frame status, so it is an exponential-complexity time cost searching problem.

Jiwen Li (li@ifi.unizh.ch), Christoph Kiefer (kiefer@ifi.unizh.ch)

Optimality:
Depth-First Search is a brute-force searching strategy which only uses the predefined information of initial status, goal status and actions, there is no use of any heuristic function as evaluation function to guide the search. So this strategy can only guarantee to find a solution for the solvable block puzzle problems.

3. Iterative Deepening Depth-First Search

Completeness:
This strategy is a variation of the standard Depth-First Search. It finds the best depth limit during the depth-first searching. So it definitely finds a solution for a solvable block puzzle problem.

Space:
Like depth-first searching strategy, the memory cost for this strategy is modest. For a block puzzle with m possible actions and a minimum n steps to find the solution, the total number of frame status in memory is about O(m*n), so it is also a linear-complexity memory cost searching problem.

Time:
Compared with the standard depth-first searching strategy, this strategy saves time in worst case. Let us define b as the number of actions, d is the minimum steps to find the solution, m is the maximum steps to find the solution, here d <= m. Depth-first search time cost is about O(b^m), iterative deepening depth-first search time cost is d*b + (d-1)*b^2 + … + b^d = O(b^d). Compared with the standard breadth-first searching strategy, this strategy saves time either. Breadth-first search time cost is b + b^2 + … + b^d + (b^(d+1) – b) = O(b^(d+1)).

Optimality:
Iterative deepening depth-first search is also a brute-force search strategy which only uses the predefined information of initial status, goal status and actions. There is no use of any heuristic function as evaluation function to guide the search. However, by finding the shallowest depth limit, this strategy can find an optimal solution for a solvable block puzzle problem.

Jiwen Li (li@ifi.unizh.ch), Christoph Kiefer (kiefer@ifi.unizh.ch)

25. Nov. 2004

4. Greedy best-first search

Completeness:
This strategy resembles depth-first search in the way it prefers to follow a single path all the way to the goal, but will back up when it hits a dead end. It arranges the searching order by using the heuristic function. As in our strategy design, we use the distance between current status and the goal status as our heuristic function. So in worst case, it behaves similar as the normal depth-first searching strategy, thus, it definitely finds a solution for a solvable block puzzle problem.

Space:
Greedy best-first search store all expanded status in memory, so the memory cost is disastrous in worst case. For a block puzzle with b actions and a maximum of n steps to find the solution, the memory cost is O(b^n), so it is a exponential-complexity memory cost searching problem in worst case.

Time:
The efficiency of greedy best-first search is dependent on the design of the heuristic function, a very good function will find the solution in linear time cost. In worst case, the time complexity is about O(b^n), where b is the number of actions and n is the maximum number of steps to find the solution.

Optimality:
Greedy best-first search can not guarantee to find the optimal solution for a solvable block puzzle. But it may dramatically reduce the algorithm time cost, if we design a good heuristic function.

Jiwen Li (li@ifi.unizh.ch), Christoph Kiefer (kiefer@ifi.unizh.ch)

5. A search*

Completeness and Optimality:
In principle, if h(n) satisfies both admissible heuristic and consistency, the A* search is both complete and optimal. It arranges the searching order by considering the heuristic function of estimated cost to the goal and the cost from initial status to the current status. As in our strategy design, we use the distance between the current status and the goal status as our heuristic function h(n). We use the minimum traveling steps between the current status and the initial status as our heuristic function g(n). By defining h(n) this way, h(n) satisfies both admission and consistency, so A* search definitely finds an optimal solution.

Space and Time:
Because A* search stores all expanded statuses in memory, the efficiency of A* search is dependent on the design of the heuristic function. A heuristic which satisfy both admissible and consistent requirement will find the optimal solution straightforward, the corresponding time and space cost could be in linear complexity. However, an unsatisfied heuristic function will take the time and space cost back to exponential complexity.

Jiwen Li (li@ifi.unizh.ch), Christoph Kiefer (kiefer@ifi.unizh.ch)