Reissuing of A.I. Take-Home Final Exam Question from Summer 2012
Description of Robotics Club Upper-Level Logic
Author: Sam Gould
Date: 2/20/13

Preview of document topics

- Problem description
- Terminology
- Representation of physical grid-world objects in robot's program memory
- Initialization of data structures
- High-level sensor interaction
- Mutation of data structures
- ASCII test grid
- Return-home algorithm
  - Pseudo-code
  - Example of algorithm in action:
    - ASCII grid representation of state
    - Diagram of the Search Tree the robot generates while performing algorithm
  - No code for return-home algorithm in this document

Note

- The return-home algorithm may be used between any two nodes on grid (so it may be applicable for any node-to-node traversal where at the robot has visited both the start and end node at least once, so as to obtain the necessary environmental information between them on which to base the search)

INTRODUCTION

Background

The university robotics club has entered a competition. The goal is to create a robot that autonomously finds a red target within a grid of floor tiles with obstacles in the way. The robot starts in the upper-left-most floor tile and blindly traverses the grid, searching for the target. Once the target is found, the robot must sing and/or dance and then return to the upper-left-most floor tile. The team to find the target and return to the starting space the fastest is the victor. The robot may *not* push obstacles out of the way or move outside of the grid at any time.

Problem Description

The robotics club team members have already implemented an efficient algorithm for finding the target floor tile, but they still need to develop an algorithm for returning the robot to the starting space. Sensor reads collected and stored from the journey to the target floor tile may be used in determining the fastest route home.

Performance Measure

The performance measure that defines degree of success is based on the number of floor tiles traversed on the robot’s journey home. The best path is the one containing the fewest number of tiles and is not obstructed by any obstacles.

Proposed Solution

My proposal for implementing an algorithm for finding the optimal route home would be to use a branch-and-bound search with dynamic programming. The initial search tree would be based on the knowledge that the robot collected through sensors on its way to the target. As the robot makes its way back to the starting space, it may update this search tree based on new environmental information it finds. The full pseudo-code algorithm is given at the end of the problem.

All code examples given in the following writing were written completely by me. Members of robotics club can vouch for this fact.

AGENT DESCRIPTION

Agent Type

- Autonomous robot navigation system
  - Agent that keeps track of the world
  - Goal-based agent
  - Rational, rather than omnipotent (does not know everything about environment)

Environment

- 7x6 grid of floor tiles (a.k.a. “grid world”)
- Tiles are covered in black vinyl tape
- White vinyl tape covers the boundaries between tiles
- Wooden obstacles are randomly placed over boundaries
- Obstacles may fully or partially separate two adjacent floor tiles
- Obstacles may only exist over boundaries

Goals

- Find the fastest route to the upper-left-most floor tile in the grid from the current location using the following information:
  - Data collected from initial journey from the upper-left-most tile to current location
  - Additional data to be collected on journey from current location to upper-left-most tile

Percepts

- Photocell on bottom-left to sense white vinyl tape on boundaries to the left-
- Photocell on bottom-right to sense white vinyl tape
- Bump sensor on left to sense blocks to the left
- Bump sensor on right to sense blocks to the right
- Bump sensor on front-right to sense blocks to front-right
- Bump sensor on front-left to sense blocks to front-left

Actions

- Move forward one floor tile
- Turn left 90 degrees
- Turn right 90 degrees

Terminology

Common Word or Phrase	Technical Synonym
Floor tile	Node
Boundary between two floor tiles	Segment
Obstacle	Block

SYSTEM DESCRIPTION

What the Robot Knows About the Environment

The following code, written in C, shows the persistent data structures that the robot uses to describe its environment. These data structures are updated when the robot moves and reads sensors. They are referenced when the agent wants to make an intelligent decision for planning its movement.

```
/* Constants: */
/* ========== */
#define NUM_ROWS 7    /* number of rows in grid-world */
#define NUM_COLS 6    /* number of columns in grid-world */
#define NUM_NODES 42  /* 7 * 6 = 42 */


/* ADT's: */
/* ====== */
typedef enum {UP = 0, DOWN = 1, LEFT = 2, RIGHT = 3, FRONT = 0, BACK = 1} Direction;
typedef enum {FALSE, TRUE} Bool;                 /* ANSI C does not have a primitive Bool type */
typedef enum {UNVISITED, VISITED} Node;          /* indicates whether or not a node has been visited */
typedef enum {BLOCKED, UNBLOCKED, IDK} Segment;  /* indicates knowledge of presense of blocks */


/* Position and Orientation of Robot: */
/* ================================== */
int current[2];       /* [0]: current row, [1]: current column -- updates in moveForward() */
Direction direction;  /* stores current direction -- updates in turn functions */

/* Environmental Information: */
/* ========================== */
Node grid[NUM_ROWS][NUM_COLS];          /* 2D array comprised of every Node in grid;
                                           UNVISITED or VISITED -- updates in moveForward() */
Segment horizSeg[NUM_ROWS+1][NUM_COLS]; /* 2D array comprised of all horizontal Segments;
                                           BLOCKED, UNBLOCKED, or IDK -- updates in checkForBlocks() */
Segment vertSeg[NUM_ROWS][NUM_COLS+1];  /* 2D array comprised of all vertical Segments;
                                           BLOCKED, UNBLOCKED, or IDK -- updates in checkForBlocks() */
```

Actions That the Agent Can Perform

The robot can move either **UP**, **DOWN**, **LEFT**, or **RIGHT** if the adjacent tile in that direction is **UNBLOCKED**.

Percept Sequence

The “percept sequence” is everything that the agent has perceived so far. At the very beginning of the grid-world navigation (i.e. when the robot starts out trying to find the target node), the robot does not know where any of the obstacles are placed in the grid. Therefore, it can initialize its **horizSeg** and **vertSeg** arrays with **IDK**, which stands for “I don’t know”. However, it does know that it cannot exit the grid at any time during its navigation, so it establishes imaginary blocks on the outer edges of the grid. The initialization sequence for the Segments is as follows:

```
/* initialize horizontal segments */
for (i = 0; i < NUM_ROWS + 1; ++i) /* set all horizontal segments to IDK */
{
        for (j = 0; j < NUM_COLS; ++j)
                horizSeg[i][j] = IDK;
}
for (i = 0; i < NUM_COLS; ++i)     /* set border horizontal segments to BLOCKED */
{
        horizSeg[0][i] = BLOCKED;
        horizSeg[NUM_ROWS][i] = BLOCKED;
}

/* initialize vertical segments */
for (i = 0; i < NUM_ROWS; ++i)     /* set all vertical segments to IDK */
{
        for (j = 0; j < NUM_COLS + 1; ++j)
                vertSeg[i][j] = IDK;
}
for (i = 0; i < NUM_ROWS; ++i)     /* set border vertical segments to BLOCKED */
{
        vertSeg[i][0] = BLOCKED;
        vertSeg[i][NUM_ROWS - 1] = BLOCKED;
}
```

Also at the very beginning of the navigation to find the target node, the robot should also initialize its current orientation and position as well as all of the grid nodes:

```
int i, j;  /* loop control variables */

/* current orientation and position of robot */
direction = DOWN;                /* robot starts by facing DOWN in the grid */
current[0] = 0; current[1] = 0;  /* top-left corner */

/* initialize all grid Nodes to UNVISITED, except for the starting Node */
for (i = 0; i < NUM_ROWS; ++i)
{
        for (j = 0; j < NUM_COLS; ++j)
        grid[i][j] = UNVISITED;
}
grid[0][0] = VISITED;            /* top-left corner */
```

As the robot moves, it updates its **current** array to its current position within the grid. Also, if it enters a node that it has not visited before, it senses the surrounding segments for blocks, updates the **horizSeg** and **vertSeg** arrays at the positions surrounding the node with **UNBLOCKED** or **BLOCKED** (depending on whether a block was sensed or not), and sets the current node to **VISITED**. The code for these operations is shown below:

```
/* update current array */
if (direction == UP)
        current[0]--; /* decrement row */
else if (direction == DOWN)
        current[0]++; /* increment row */
else if (direction == LEFT)
        current[1]--; /* decrement column */
else /* direction == RIGHT */
        current[1]++; /* increment column */

/* if the node has not been visited before or the node is a corner node,
   scan for blocks and set node in grid to VISITED */
if (grid[current[0]][current[1]] == UNVISITED || isCorner() == TRUE)
{
        checkForBlocks();
        grid[current[0]][current[1]] = VISITED;
}
```

```
/* if adjacent Segments are not BLOCKED (either IDK or UNBLOCKED), sensor functions are called to
   sense surrounding segments (UNBLOCKED Segments are tested to increase our chances of not
   missing a block due to a faulty sensor read) */
void checkForBlocks(void)
{
        /* define aliases for adjacent segments */
        Segment *top, *btm, *lft, *rgt;
        top = &(horizSeg[current[0]][current[1]]);
        btm = &(horizSeg[current[0] + 1][current[1]]);
        lft = &(vertSeg[current[0]][current[1]]);
        rgt = &(vertSeg[current[0]][current[1] + 1]);
        if (direction == UP)
        {
                if (*top != BLOCKED)
                {
                        if (frontL_s() == TRUE || frontR_s() == TRUE)
                                *top = BLOCKED;
                        else
                                *top = UNBLOCKED;
                }
                if (*lft != BLOCKED)
                {
                        if (left_s() == TRUE)
                                *lft = BLOCKED;
                        else
                                *lft = UNBLOCKED;
                }
                if (*rgt != BLOCKED)
                {
                        if (right_s() == TRUE)
                                *rgt = BLOCKED;
                        else
                                *rgt = UNBLOCKED;
                }
        }
        else if (direction == DOWN)
        {
                if (*btm != BLOCKED)
                {
                        if (frontL_s() == TRUE || frontR_s() == TRUE)
                                *btm = BLOCKED;
                        else
                                *btm = UNBLOCKED;
                }
                if (*rgt != BLOCKED)
                {
                        if (left_s() == TRUE)
                                *rgt = BLOCKED;
                        else
                                *rgt = UNBLOCKED;
                }
                if (*lft != BLOCKED)
                {
                        if (right_s() == TRUE)
                                *lft = BLOCKED;
                        else
                                *lft = UNBLOCKED;
                }
        }
        else if (direction == LEFT)
        {
                if (*lft != BLOCKED)
                {
                        if (frontL_s() == TRUE || frontR_s() == TRUE)
                                *lft = BLOCKED;
                        else
                                *lft = UNBLOCKED;
                }
```

```
                if (*btm != BLOCKED)
                {
                        if (left_s() == TRUE)
                                *btm = BLOCKED;
                        else
                                *btm = UNBLOCKED;
                }
                if (*top != BLOCKED)
                {
                        if (right_s() == TRUE)
                                *top = BLOCKED;
                        else
                                *top = UNBLOCKED;
                }
        }
        else /* direction == RIGHT */
        {
                if (*rgt != BLOCKED)
                {
                        if (frontL_s() == TRUE || frontR_s() == TRUE)
                                *rgt = BLOCKED;
                        else
                                *rgt = UNBLOCKED;
                }
                if (*top != BLOCKED)
                {
                        if (left_s() == TRUE)
                                *top = BLOCKED;
                        else
                                *top = UNBLOCKED;
                }
                if (*btm != BLOCKED)
                {
                        if (right_s() == TRUE)
                                *btm = BLOCKED;
                        else
                                *btm = UNBLOCKED;
                }
        }
        return;
}
```

At the beginning of the algorithm for returning home, the robot knows about its environment from sensor readings on its journey to its current position within the grid. This knowledge is stored in the **horizSeg** and **vertSeg** arrays in the form of **UNBLOCKED** and **BLOCKED** values. The robot also knows which nodes it has visited and which nodes it has not visited. This information is stored in the **grid** array in the form of **VISITED** and **UNVISITED** values.

FULL SOLUTION PROPOSAL

Graphical Representation of Grid

In order to view the intelligence of the robot while isolating it from the “fuzzy” nature of the real world, I devised an ASCII grid system to represent the actual grid-world. This also helps to visualize the intelligence and action of the robot to help devise a solution for the fastest-route-home problem. An example grid is shown below, after the very important “Key” section for the indicators on the grid.

Key:
Robot direction indicators:

```
  ^
<   >
  v
```

Obstacle indicators:

	Sensed?	Blocked?
? / ???	no	no
| / ---	yes	no
B / BBB	no	yes
# / ###	yes	yes

Other indicators:

.	VISITED node
R	target node

Grid-world representation when starting to find the target node:

```
 +###+###+###+###+###+###+
0# v ?   ?   ?   B   ?   #
 +???+???+???+???+???+???+
1#   ?   ?   ?   B   ?   #
 +BBB+???+???+???+???+???+
2#   ?   ?   ?   ?   B   #
 +???+???+???+BBB+???+???+
3#   ? R ?   ?   ?   ?   #
 +???+???+???+???+BBB+???+
4#   ?   ?   ?   ?   ?   #
 +???+???+???+???+???+???+
5#   ?   ?   ?   ?   B   #
 +???+???+???+???+???+???+
6#   ?   ?   ?   ?   B   #
 +###+###+###+###+###+###+
```

The target node does not always have to be at (3, 1). It was just arbitrarily placed there in this example. In the following grid, we can see the state of the environment after the robot has found the target node:

```
 +###+###+###+###+###+###+
0# . |   ?   ?   B   ?   #
 +---+---+---+---+???+???+
1# . | . | . | . #   ?   #
 +###+---+---+---+---+???+
2#   ?   ?   | . | . #   #
 +???+---+---+###+---+???+
3#   | < | . | . | . |   #
 +???+---+---+---+###+???+
4#   ?   ?   ?   ?   ?   #
 +???+???+???+???+???+???+
5#   ?   ?   ?   ?   B   #
 +???+???+???+???+???+???+
6#   ?   ?   ?   ?   B   #
 +###+###+###+###+###+###+
```

Input

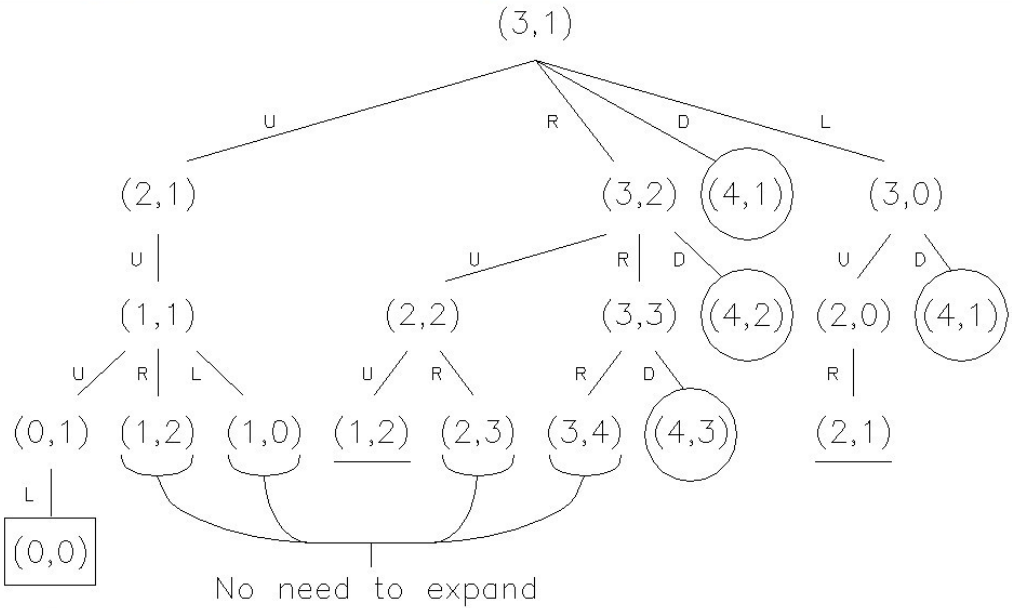
Realizing that the problem at hand is for the robot to find its way back to the starting node, the agent's input is the information stored in the **horizSeg** and **vertSeg** arrays—this is what the agent currently knows about the segments on the grid. One may be tempted to think that the **grid** array would be good input information, but it is not. The reason is that even if the robot has visited a node in the grid, the node is not guaranteed to be accessible from all directions. Therefore, the segments are the appropriate input.

Processing

As stated before, a branch-and-bound search with dynamic programming will be used to find the fastest way back to the starting node in the grid coordinate (0, 0). The pseudo-code algorithm is given below:

```
1) Form a one-element queue consisting of a zero-length path that
   contains only the root node
2) Until the first path in the queue terminates at node (0, 0) in
   the grid or the queue is empty,
      a. Remove the first path from the queue; create new paths by
         extending the first path to all the neighboring nodes whose
         entry-side segment is UNBLOCKED; attempt to expand in the
         following order: UP, RIGHT, DOWN, LEFT
      b. Reject all new paths with loops
      c. Add the remaining new paths, if any, to the queue
      d. If two or more paths reach a common node, delete all those paths except
         the one that reaches the common node with the minimum cost; if there are
         two or more paths that reach the same node at the same travel cost,
         discard all but the first path in that group
3) If the goal node is found, announce success; otherwise, announce failure
```

The following tree illustrates the result of the search when applied to the example grid in the "Graphical Representation of Grid" section. (x, y) represents nodes on the grid, with x being the row number and y being the column number. The "U", "R", "D", and "L" symbols on the links indicate the direction that the search was expanded (up, right, down, and left). The circles around nodes mean that they cannot be further expanded. The underlined nodes are those whose path's cost is greater-than or equal-to an existing path, so it is trimmed from the search tree. The node with the rectangle around it indicates is the start node (where the agent wants to go).



The robot starts at node (3, 1). Although it has not visited node (2, 1), the agent knows that it is accessible from node (3, 1) because it has sensed the segment between the two nodes to be **UNBLOCKED**. Because the robot has not visited node (2, 1) or the node to the **LEFT** or **RIGHT** of it, it does not know if the segments are **BLOCKED** or **UNBLOCKED**—in other words, those segments contain values of **IDK** in the robot's memory (this appears as '?' on the grid). Therefore, when the agent expands its search at that node, it cannot expand to the **LEFT** or **RIGHT** directions, but it can expand **UP**. It does this, and finds that this "shortcut" through an **UNVISITED** node has quickly placed it close to node (0, 0). Once the agent has found the optimal path using the given information, there is no need to expand the other nodes because they are all on the same level of the tree and the links all contain the same weight.

Output

The output of the algorithm is the optimal path to the starting node, based on what the agent knows. As the robot follows this path to the home space, it may pass through nodes that it has not visited before.

As the robot enters a previously **UNVISITED** node, it calls the **checkForBlocks** function and updates its memory. Because the robot has more available information, it needs to start a new dynamic branch-and-bound search from its current position. This new search will then return the new optimal path, which the robot will follow. This process continues until the robot's current position is (0, 0). In the above example, the agent did perform a new search at (2, 1), but it did not find a faster path than was originally intended, so the robot continues to follow the path it is/was on.