Software Engineering Project

- Bug World Description -

P. Baumann, Constructor University, Spring 2023

Bug World

This document describes the Bug World game engine. Bug World is an environment where the behavior of swarms of bugs can be observed (see Figure). Goal of the bugs is to collect food and bring it home. There are competitive swarms – called the red and the black swarm for distinction – having the same goal, but with their own home. The best swarm is the one with the most food in its home base after a given amount of time (i.e., simulation steps). Hunting for food is competitive, and bugs may kill each other.

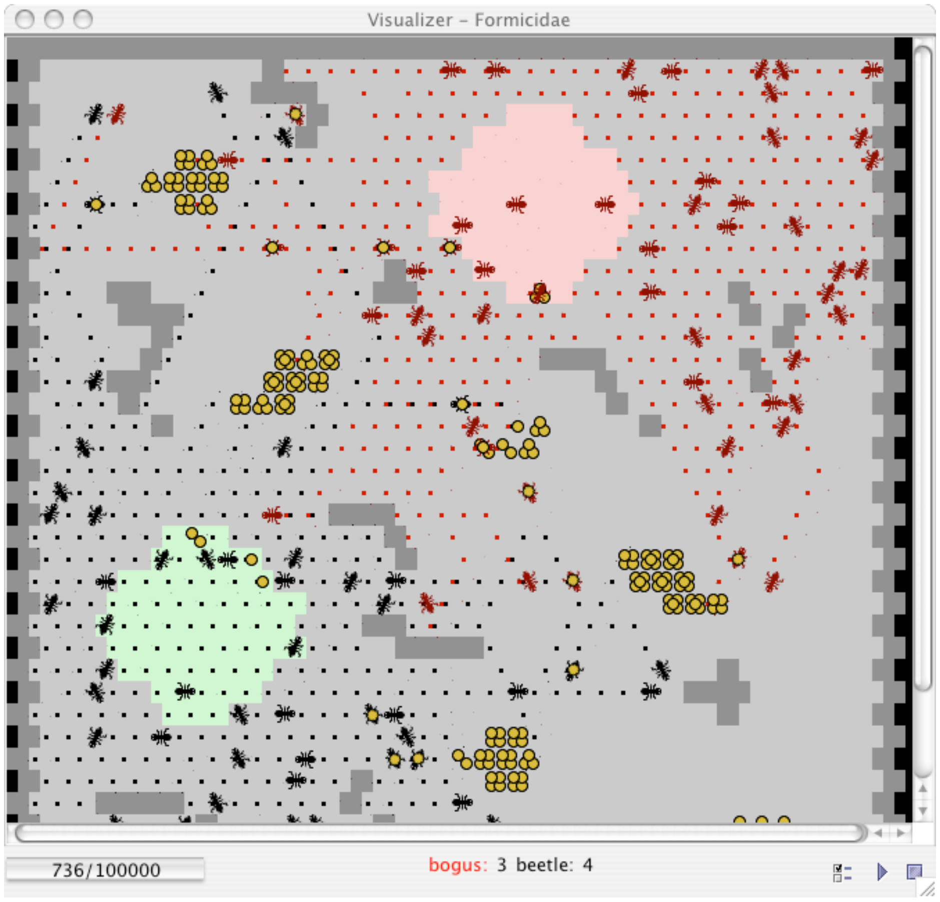

Figure 1: sample world map with red and black bug cohorts, obstacles, and food.

A further complexity is given by the landscape which can be set with each experiment (simulation run) which can have different shapes and may contain obstacles. Hence, a world (also called environment) consists of cells which can contain obstacles, food, and bugs. The home base (nest) of a swarm is a dedicated contiguous area inside the world.

A bug brain is a finite state automaton controlling the bug behavior. All bugs of a swarm share the same brain, described as a bug program code. The bug behavior can be guided by the individual bug's situation, and so despite the common brain the bugs of a swarm will develop individual behavior. Additionally, bugs may leave traces in cells indicating that one of their swarm was in this cell. These traces coordinate swarm behavior, but they are readable by adversarial bugs as well.

The world simulation proceeds in ticks, up to a limit provided. With each tick, one instruction from each bug code is executed, thereby updating the bug states as well as the world state accordingly.

Each experiment, called a tournament, consists of two runs where two codes compete against each other. For the second run, the starting position of red and black are swapped so that no bug code can take advantage of a particular landscape.

A bug world is a 2D flat map containing hexagonal cells, each identified through its (x,y) position (see Figure). The top-left corner cell has position (0,0), coordinates increase to bottom and right. Coordinates are natural numbers.

Adjacency is defined as follows. Let p=(x,y) be a position and d a direction. Then, **adjacent(p,d)** delivers the cell immediately neighbouring p in direction d. See Figure above for the encoding of directions.

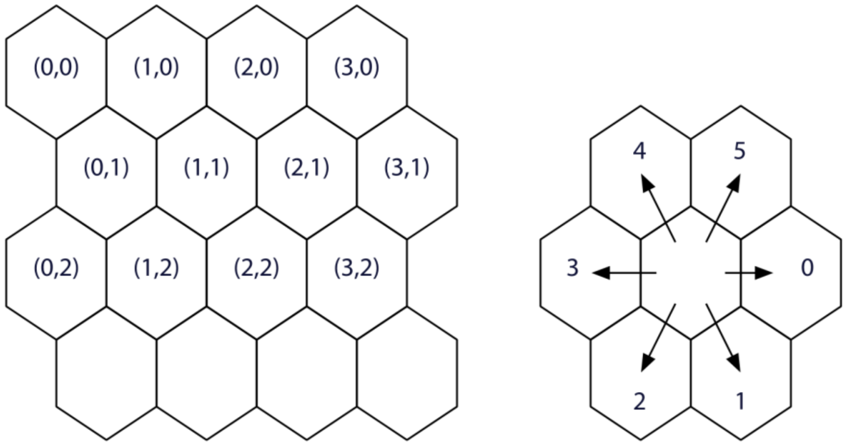

Figure 2: hexagonal world layout

A cell is either free (ie, bugs can sit there) or it is obstructed. Predicate **obstructed(p)** returns true if position p is obstructed. A free cell can simultaneously contain:

- One bug at most
- A non-negative number of food packages

- Markers from each swarm.

Several functions serve to read and set a cell state:

- **Occupied(pos):** true if cell at pos contains a bug
- **Bug_at(pos):** returns the bug sitting at position pos
- **Place(pos,b):** assign bug b to position pos
- **Free(pos):** clear cell at position pos from bugs

A cell can contain any number of food packages. A bug can carry at most one food package at a time. These functions manage food:

- **food_at (pos):** return the number of food units in the cell at position pos
- **set_food_at(pos,n):** set cell at position pos to hold n food units

Each bug swarm has a homebase, its nest. Corresponding functions are:

- **base_at(pos,c):** returns true if cell at position pos is part of the nest of the swarm with color c
- **other_base_at (pos,c):** returns true of cell at pos belongs to the adversarial nest

World

Overall Layout

World defines a 2-dimensional world geometry. An absolute direction in this world is encoded as an integer, which is not pretty but simplifies module dependencies. A relative direction is a turn. While not apparent from the interface, in this particular world each position has six neighbors (see Figure 2).

Bugs

Overview

Two types of bugs coexist in the world, red and black. Function **other_color()** determines the enemy color. The internal bug state is given by the following components:

- **Id**: A unique positive number. It determines the evaluation sequence in the bug swarm for each simulation tick.
- **Color**: the bug's color
- **State**: a non-negative integer number representing the bug's internal state (remember, while all bugs follow the same program they have their individual state)
- **Resting**: duration of a sleep – after a successful move the bug sleeps for 14 cycles during which resting gets decremented until the bug can act again
- **Direction**: current absolute heading
- **Has_food**: a Boolean value indicating whether the bug is currently carrying food. A bug can carry only one food item at a time.

A bug can change its direction (i.e., turn) left or right relative to its current heading. This determines the new heading through functions left(d) = (dir + 5) mod 6 and right(d) = (dir + 1) mod 6.

A bug can also investigate its immediate neighbourhood, concretely: its own cell, the cell ahead, left ahead, and right ahead. Function sensed_cell(p,dir) determines the target position from position p and absolute heading dir.

Bugs can set 6 markers into different cells. Each marker is identified by a number ranging from 0 to 5. Every swarm has its individual set of 6 possible markers which can be handled with the following functions:

- **Set_marker_at (pos,c,i):** set marker I at position pos for swarm of color c
- **Clear_marker_at(pos,c,i):** delete marker i at position pos for swarm of color c
- **Check_marker_at(pos,c,i):** return true of marker i is set at position p for color c
- **Check_other_marker_at(pos,c,i):** return true if at position pos any marker is set by the adversarial swarm

Before the first iteration the world contains no markers.

Bugs can differentiate own markers for their number, for adversarial markers they only can sense their presence, but not their value.

A bug can sense its environment for 10 parameters:

- Friend: cell is occupied by bug of own color
- Foe: cell is occupied by adversarial bug
- FriendWithFood: cell is occupied by friendly bug carrying food
- FoeWithFood: cell is occupied by adversarial bug carrying food
- Food: cell contains food
- Rock: cell is an obstacle
- Marker i: cell contains own marker with number i
- FoeMarker: cell has adversarial marker
- Home: cell is part of own nest
- FoeHome: cell is part of adversarial nest

Predicate **cell_matches(pos,cond,c)** returns true if position pos fulfils condition cond (one of the 10 choices above) for color c.

Some more bug management functions:

- **Dead(b):** set bug b state to dead
- **Position(b):** return position of bug b
- **Kill(pos):** kill bug at position pos and remove it from the corresponding cell

Bug Brains

A bug brain is given as a sequence of instructions taken from the Bug World opcodes. Instructions are expressed as bytes so that a complete bug program is a binary string which can be stored conveniently in a file. However, it is not convenient to read, and therefore a bug assembler language is available allowing describing bug programs on a higher, more human-readable level. A Bug World component, called assembler, translates this language into machine instructions.

Bug Assembler Grammar

Each instruction has access to the bug's state and the cell on which the bug sits currently. This is the complete instruction set:

sense sensedir s1 s2 cond	check if condition cond is fulfilled in direction sensedir; if yes, go to s1, otherwise s2.
mark m s	set marker m in current cell, then go to s.
unmark i s	delete marker I, then go to s.
pickup s1 s2	take food from current cell, then go to s1; if no food is available or bug already carries food then go to s2.
drop s	put food into current cell and go to s.
turn lr s	turn in direction lr (left or right), then go to s.
move s1 s2	advance by once cell in current direction, then go to s1; if cell ahead is blocked go to s2.
flip p s1 s2	obtain a random number between 0 and p – 1; if zero then go to s1 , otherwise go to s2.
direction d s1 s2	if current heading is d then go to s1 , otherwise go to s2.

From these instructions programs can be built and stored as text files. The grammar is as follows:

```
program ::= instruction+
instruction ::=
      | "Sense" dir cond "then" label "else" label
      | "Mark" int "then" label
      | "Unmark" int "then" label
      | "PickUp" "then" label "else" label
      | "Drop" next
      | "Turn" leftright
      | "Move" "then" label "else" label
      | "Flip" int "then" label "else" label
      | "Direction" int "then" label "else" label
      | string ":"
label ::= string | number
dir ::= "Here" | "LeftAhead" | "RightAhead" | "Ahead"
leftright ::= "Left" | "Right"
cond ::=
      | "Friend"
      | "Foe"
      | “FriendWithFood”
      | “FoeWithFood”
      | “Food”
      | “Rock”
      | “Marker” int
      | “FoeMarker”
      | “Home”
      | “FoeHome”
```

The last instruction variant is not strictly an instruction that gets executed, but associates a name with the current position in the code; this can be used as a target for goto as a more coding friendly alternative to the (numerical) absolute instruction addresses.

Symbol classes: Important are names, in particular those for comments, numbers, identifiers, and what constitutes a tile in a map. They are given by the following regular expressions (numbers are decimal):

```
number  = ['0'-'9']+
ws      = [' '|'\t'|'\n'|'\r']+
id      = ['A'-'Z'|'a'-'z']+
comment = ';' [^ '\n' '\r']*
misc    = [':']
nl      = '\n' '\r'?
```

Below an example bug code is given, resembling a “stupid bug” which randomly walks around, and if food is found the bug picks it up and likewise randomly walks further until it finds its nest an can drop the food (“;” indicates a comment, after which the instruction number is indicated to ease reading):

```
sense ahead 1 3 food         ; [ 0]
move 2 0                     ; [ 1]
pickup 8 0                   ; [ 2]
flip 3 4 5                   ; [ 3]
turn left 0                  ; [ 4]
flip 2 6 7                   ; [ 5]
turn right 0                 ; [ 6]
move 0 3                     ; [ 7]
sense ahead 9 11 home        ; [ 8]
move 10 8                    ; [ 9]
drop 0                       ; [ 10]
flip 3 12 13                 ; [ 11]
turn left 8                  ; [ 12]
flip 2 14 15                 ; [ 13]
turn right 8                 ; [ 14]
move 8 11                    ; [ 15]
```

Programmatically, a bug is just an object containing a collection of values, some of them immutable. When created by **create(color,prg,pos,id)**, a bug's **color**, program **prg**, initial position **pos** (mutable) and identity **id** are defined. Initially, a bug is facing into direction 0, carries no food, is in state 0, and needs no resting (0).

Observers:

- **color**: A bug has a color. By design we don't need to know how many colors there are and a color could be abstract if we had a generator to generate different colors.
- **state**: current state
- **resting**: wait cycles
- **direction**: facing into this direction
- **has_food**: carrying a bit?
- **position**: actual position
- **id**: ID
- **instr**: current instruction, depends on state
- **to_string**: for debugging

Mutators update the internal mutable state:

- **set_state**
- **set_resting**
- **set_direction**
- **set_has_food**
- **set_position**

Bug Machine Code Instructions

An instruction is directly executable by a bug (i.e., the simulator), so we can consider it the "machine code" or "executable code" in analogy to CPUs. A program is a sequence of instructions where each instruction is addressed by its index in the sequence. For the semantics of instructions refer to the Engine module.

All the opcodes (such as "Move") and the symbolic constants used (such as "FoeHome") need to be mapped to single-byte numbers. States require two bytes to allow for more than 255 states.

Implementation Architecture

Overview

The Bug World game runs entirely in a Web browser. It can, on a high level, be subdivided into the GUI (screen and interaction management), assembler (generating an executable bug program from assembler input), simulator (the engine executing the bug codes and updating its world state), plus auxiliary functionality. The latter includes

- For debugging, a **to_string()** function with every class which outputs the current internal object state in a human-readable format.
- Function **rand(n)** returns a pseudo-random integer in the range 0, and n − 1 inclusive.

GUI

The GUI component manages the screen, accepting user input and displaying responses. For doing so it internally invokes the further modules, in particular: the assembler and the simulator.

- User input:
    - A world map file
    - Two bug assembler source code files, one for the red and one for the black bug; instead of calling the competing bugs red and black, individual names may be provided (eg, by deducing them from the code file names)
    - Number of iterations the simulation should run
    - Optional controls, such as activating log output
- Output:
    - A visual representation of the world map, updated with every simulation step
    - The current iteration counter, and how many iterations are left
    - A summary of the current tournament status (amount of undetected food; for red and black bugs: food brought home, bugs killed / remaining; etc.)
    - If enabled: log output on screen

In this setup, the assembler code gets translated into bug machine code which then is run in the simulator (i.e., interpreter).

Further, there should be a means to just load one bug assembler program, show the machine code generated (if error free) or errors found otherwise.

Simulator Top Level

This is the simulator toplevel: it parses the command line, provides defaults, and holds everything together. The idea is to have none of the truly algorithmic aspects in this module.

Engine

This module defines the operational semantics of the bug machine language and this is the heart of the simulator. A simulator is created from three files (or file handles) for the world, red bugs, and black bugs respectively. When a simulator t is run it responds with statistics about the state of the word at the end of the simulation. The simulation lasts for n cycles; every cycle the simulator is logged if logging is activated.

Tournament

This module provides a tournament for bugs: all pairs of bugs compete twice in a given environment. A winning bug receives 2 points, a draw 1 point, a loser 0 points. In the second competition for a given pair the roles of black and red bugs are reversed.

The run function receives file names for the environment world and the bugs, characterized by their machine instructions. The result is the number of points each bug has won in the tournament.

Worlds

Overview

A world at large is defined as a contiguous set of cells. The world as such is immutable: the content of a certain position can be read, but not overwritten. Cells may either be mutable (changing its contents in the course of the simulation) or an immutable obstruction which cannot be stepped upon by bugs.

The geometry of the world is encoded into the following three functions:

- **turn(lr,d)** is the absolute direction after looking in direction d and taking a turn lr.
- **adjacent(p,d)** is the adjacent position when looking from p into direction d,
- **sensed_cell(p,d)** where is the cell examined relative to position p, when looking into direction d and using where to select a cell.

We provide three constructor functions for the three types of cells in the initial state of a world: a cell that is part of a homebase (holding a bug), an empty cell, and an empty cell carrying some food.

Cell World

This module provides a world populated with bugs; it constitutes the central module of the simulator.

A world is built from a file that contains the world's map and two factories for red and black bugs. A factory receives a position and an id when it is asked to create a bug.

The following functions observe the geometry of the world: turns are translated into direction and neighbors into positions.

Obervers: Functions to observe the state of a world t.

- **size** returns the world's dimension in dimensions x and y.
- **obstructed** is true if a position is an obstruction and thus cannot carry anything.
- **occupied** is true if it is holding a bug.
- **bug** at returns the bug of an occupied position. It is a checked run-time error to apply bug at to a cell an unoccupied cell.
- **food at** returns the amount of food at a position.
- **base at** is true if a position belongs to bug of a given color.
- **other base at** is true if a base of a different color is at a position.
- **check marker at** is true if a position holds markers for a given color of bugs.
- **check other marker at** is true if a position holds any marker for a color different than the given one.
- **adjacent other bugs** returns the number of different-colored bugs adjacent to a given position.
- **cell matches** if a cell matches a condition cond for a bug of color.

Mutators: Functions to change the state of the world. It is a checked run-time error for all these functions to apply them to an obstructed position.

- **set food at places** food (bits) at a position.
- **free** removes a bug from a position; this does not affect the liveness of the bug.
- **set marker at** sets a marker i for bugs of a certain color.
- **clear marker** at removes a marker i for bugs of a certain color.
- **place at** places a bug at a position.

Logs and Statistics: For debugging, the state of the world can be written to a log file; two formats are available: icfp log and sopra log. The latter is more compact and column-oriented, the former more verbose but they both contain essentially the same information. The stats function returns a record of statistics per bug colony: the number of alive bugs, and the amount of food (e.g. bits) placed on the colony's base.

Definition of Worlds

Worlds are described through maps, an ASCII representation of the world. The first two lines give the width x and height y of a map. All following y lines of length x outline the map with the following indicators per position:

#	obstacle
.	empty cell (ie, no bug)
-	empty cell, black swarm nest
+	empty cell, red swarm nest
1..9	empty cell with number of food units

The rim of a map should be set with obstacles as a natural boundary for the bug movement.

```
10
10
# # # # # # # # # #
# 9 9 . . . . 3 3 #
# 9 # . - - - - - #
# . # - - - - - - #
# . . 5 - - - - - #
# + + + + + 5 . . #
# + + + + + + # . #
# + + + + + . # 9 #
# 3 3 . . . . 9 9 #
# # # # # # # # # #
```

Figure: sample world map

A map initially does not contain bugs. Before the first iteration every nest cell is populated with a bug of the corresponding color.

Bug Assembler

Overview

The bug assembler translates .buggy assembler files to .bug machine instruction files (file name extensions are mere convention). At program start, the main function examines the command line and starts the translation. As its main service, this module provides a parser read for assembler instructions.

The assembler does two passes over the assembler source code.

- The first pass assigns addresses to labels (“address resolution”). To this end, it determines the position number of the machine instruction an assembler instruction is translated to in the second pass.
- The second step is the heart of the assembler: the translation of an assembler instruction to a machine instruction. There is only one interesting point: if an instruction is followed by a Goto, the Goto can be elimited by making the Goto’s target the successor of the instruction. If the first instruction is a Goto, we implement it using Flip.

The **assemble()** function implements the two passes: defining labels and translating the program to machine code. The assembler reads from stdin if no file name is provided and emits the translation to sdout.

For this to work we need an environment (or symbol table) for labels; the denotation of a label is a state (an integer).

The parser for assembler instructions simply evaluates a single line in each call, which is possible due to the simplicity of the grammar. Hence, the parser gets called in a loop which provides one line from the input file in turn until eof is reached.

The grammar for instructions is so simple, we don’t need a complicated parser: we simply pattern match over the list of tokens. The one exception are conditions **cond** because a condition may be comprised of one or two tokens. Luckily, the condition is the last element of a sense instruction. We thus pass the remaining list of tokens to function **cond** and let it figure it out.