CENG 561 : Artificial Intelligence

Term Project Final Report

Project Name : CleanBot
Project Designer : Volkan OKBAY
Instructor : Prof. Dr. Ferda Nur Alpaslan
Middle East Technical University / 11.01.2017

TABLE of CONTENTS

1. INTRODUCTION

Artificial intelligence problem are widely considered and branched in the field of 'robotics'. Popular problems include motion planning, making decisions, game-playing etc. Our problem can be desribed as the motion planning of a house type vacuum cleaner robot.

The main aim is to cover all the tiles in a room, with an efficient way. Only sensor robot has is a one tile proximity sensor. The problem is a common one in robot world. Although, the configuration in similar cases differ, there are some basic algorithms for core problem-solver.

This is an important task involving industrial cleaning, mine sweeping, and agricultural operations etc. [1]. By saving energy and time, it should be possible to cover all the area to fulfill primary task. The task is explained in detail in part 3.

Before handling the problem solving, actual commercial type samples are examined [2] also literature is scanned. Some observations are included in part 2. Then finally, "boustrophedon method" and "cell-decomposition algorithm" are selected main approach.

In the end, a GUI and working logic behind is implemented to execute a simulation of the actual task. We have used Clojure programming language. All the implementation information will be given in the 4th part. This report also involves results with two defined parameters, discussion and future work of the given study.

There is also a simple website is started for a visual introduction of CleanBot [7].

2. RELATED WORK

Our main problem is fairly popular and still developing one. For planning problems, the resourceful book from LaValle [3], is a great starting point. As it containt wide variety of types in planning.

In literature, there exist different approaches depending on the circumstances of given problem. An essential question at this point is on online vs. offline planning methods. For offline case, it is generally used a heuristic function provided by predefined map size, layout or an image of the room above. They all are useful in planning the next move more efficiently. When information is gathered about the room, an algorithm shall be implemented by different search or FOL methods.

A common type of offline planning method is about SLAM, namely 'Simultaneous Localization And Mapping'. An example is proposed by Strimel [1], in which 3-D graphical representations of ailes and layout of a convenience store is used to plan an efficient way of coverage in real world applications.

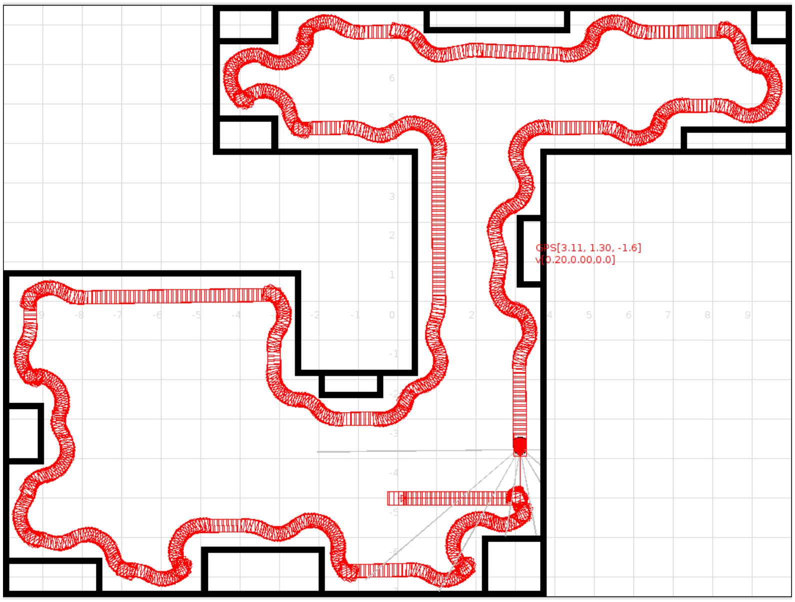

Figure 1: Wall following (*http://java-player.sourceforge.net/*)

There are also some basic types that can be used interchangeably, either in online or offline cases. One is wall following, that is used by a lot of commercial bots. As in Figure 1, the bot gets an idea about the room perimeters first.

Another simple move is random sweep. As stated in Doty et al. [6], random sweep needs some heuristics and it usually used in small ares or as a last resort where a decision is hard give (Figure 2).

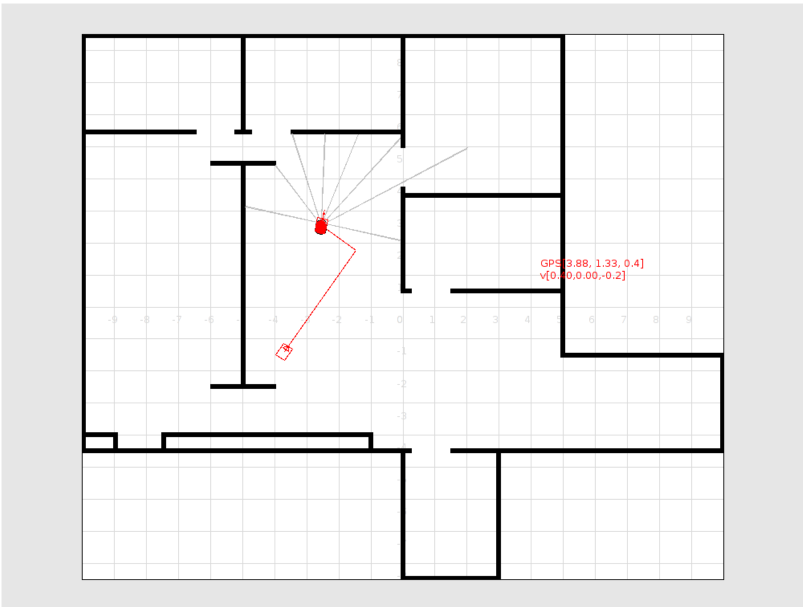

Figure 2: Random sweep (*http://java-player.sourceforge.net/*)

There also search algorithms are helpful in this problem. An example implemented Yakoubi [4], presents genetic algorithm to find the way out. In basic sense, minipaths are formed by labels of tiles. Then genetic match and crossover are done to get a productive sequence of moves.

3. PROBLEM DEFINITION and ALGORITHM

3.1 Task Definition

The task selected is in fact a planning problem in broad sense. Going down, it is examined under robot motion plannig. Aim is to navigate a robot autonomously. To be more specific, a best keyword to define our problem would be “coverage” planning. In this one, robot must be spanning all the tiles as in case for CleanBot.

For analysing problem in PEAS manner:

A- **Performance Measure :** Performance measure will be goal based. In other word, tests will ask if all the dirt on the floor is cleared and if is the current approach the least time consuming solution.

B- **Environment :** Environment consists of empty/dirty tiles, walls/obstacles and CleanBot itself. Following descriptors may be given for default features of the problem:

- Partially-observable: Room is previously unknown.
- Deterministic: Output is specified by action and current state.
- Sequential: Past and future states are important.
- Static: By default, a static environment is assumed.
- Discrete: Time, state and actions are in discrete manner.

C- **Actuators :** Actuators are movement components that can alter the location in 8-cell neighborhood (for ease of operation) and vacuumin function.

D- **Sensor :** Sensors involve a motor driver to feedback next location of CleanBot and 1 tile proximity sensor for obstacles around. Robot can move and sense tiles in the given manner as in Figure 3, left connectivity.

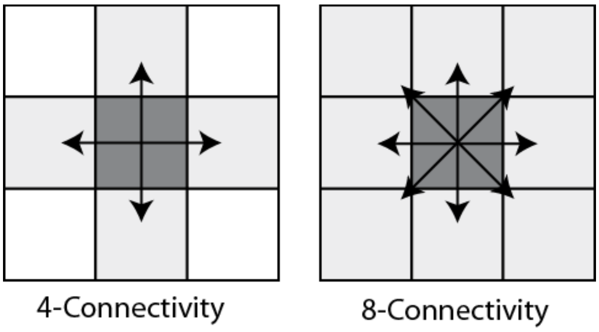

Figure 3: Pixel connectivity (https://sites.ualberta.ca/~ccwj/teaching/image/morph/)

3.2 Algorithm Definition

In this part, the methods selected for this project are explained.

The main approach is named “cell decomposition” as proposed by Choset [5]. Cell decomposition is mainly In this method, imagine a line, sweeping through the map from left to right. When a point such that connectivity of the line changes (line break or line merge), mark this point as a critical point. Those critical points would be borders of cell, which are represented with different colors in Figure 9. All critical point are marked with black dots in the Figures 4-9.

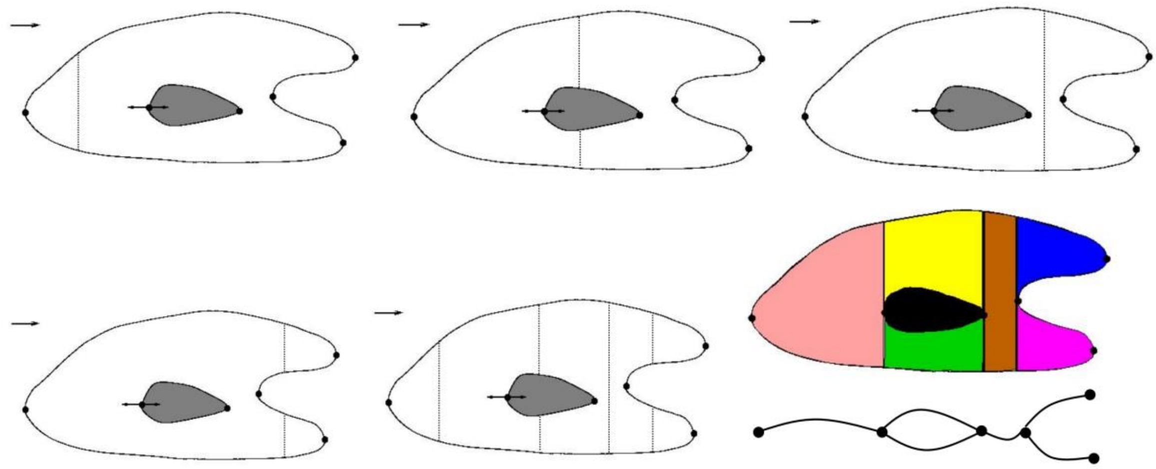

Figure 4-9: Sweep line approach in Morse Cell Decomposition (*http://www.cs.cmu.edu/*)

If this was an offline planning problem, the solution will be a sequence given by a simple Travelling Salesman Problem. As we consider online case, there needed to be a incremental line sweeping. This time, robot itself will be actual sweep line, as it travels through map. When a critical point is encountered, bot chooses the closest new cell and starst decomposing it. Incremental search is illustrated in the following Figure 10.

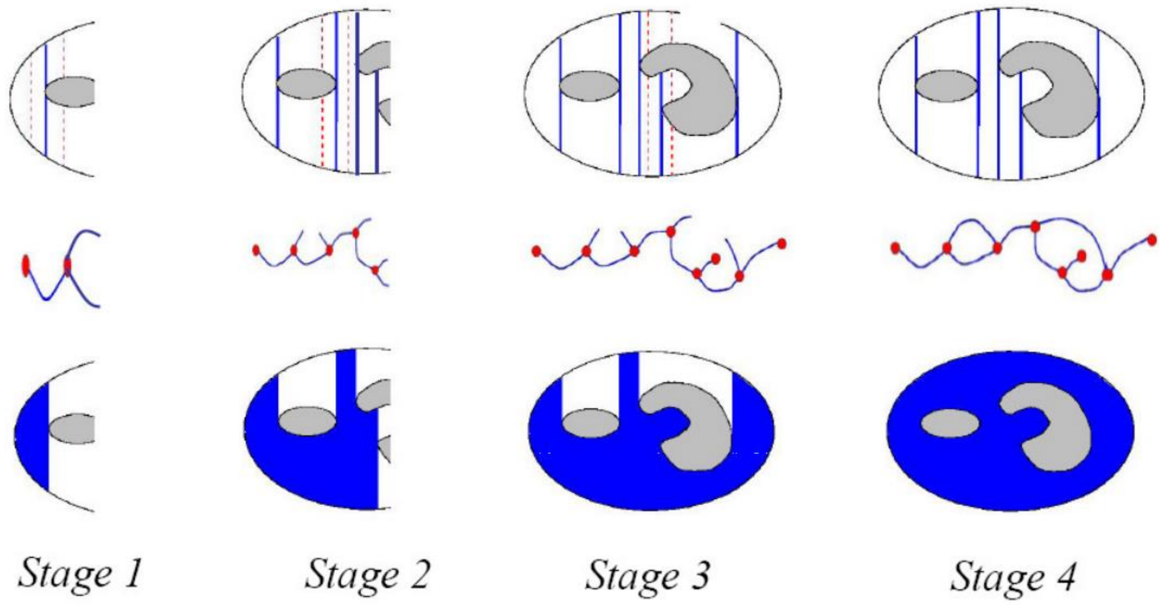

Figure 10: Incremental sweep line decomposition [5]

Notice that, when (in the stage 3) right composition is done, the robot loops back to those cells, which were not adjacent at the time of decision. So, bot needs to memorize unvisited cells if it enters to another cell.

After sweep mode, in the back-loop mode, there implemented a simple graph search to find a way from current position to the closest unvisited cell. This 2-D map graph search is done by so-called Flood Fill Algorithm [12]. Starting with current position, tiles are labeled with incrementing numbers until a new cell is flooded. By the way, only visited tiles are labeled as the robor has no idea of unvisited tiles.

R	V	V	V	R	1	V	V	R	1	2	V	R	1	2	3	R	1	2	3
V		V	F	1		V	F	1		V	F	1		3	F	1		3	4
V	V		U	V	V		U	2	V		U	2	3		U	2	3		U
U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U

Figure 11: Flood fill graph search through visited tiles

Here in Figure 11, a simple example is given on the graph search method used. The robot is located on tile R and target is tile F. Gradually give numbers to adjacent tiles as given in the figures. Note that only visited tiles are labeled. After tile F has a label, backtrack by decrementing numbers to get a list of moves.

Final method was about how the robot decomposes a single formed with tiles. The simpliest yet effective method is named after 'boustrophedon method' (Greek for ox-plow) [5]. Robot basically traverses by going up and down on the simulation screen (Figure 12-13).

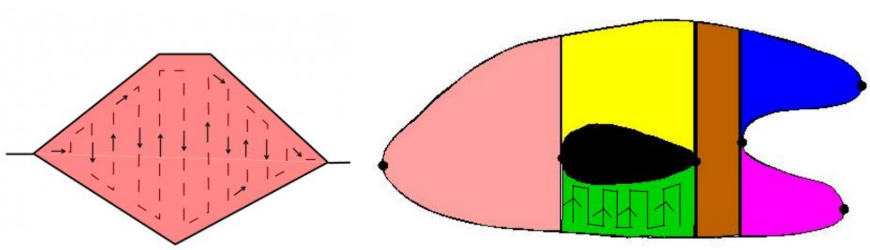

Figure 12-13: Boustrophedon decomposition (http://mdahsanhabib.weebly.com/indoor-navigation.html)

All in all, total method may be summarized as follows:

1) Start in right-sweep mode, decompose current cell.
2) If a new critical point is encountered, enter closest new cell (left cells have priority) and memorize other new cell coordinates if exist.

   Else if there is no new cell go into back-loop mode to get back to closest unvisited cell.
3) Decompose new cell. While decomposition constantly stack (only) curren left/right sweep lines.
4) If all the map is covered stop; otherwise, loop to STEP 2.

The best way to comprehend the whole method used is that running the executable simulation and simultaneously checking the status on the terminal of operating system. The status contains current map matrix representation, robot position, current sweep lines, unvisited cell coordinates to be back-looped etc.

4. EXPERIMENTAL EVALUATION

4.1 Methodology

In order to test the algorithm and methods determined, a software simulation is needed. To implement such task, there also needed a software platform to be set up. Firstly, a research is done to select a proper language, that is compatible with A.I. world and also capable of presenting a nice graphical user interface. Among choices like Prolog, Common LISP, Scheme, Rackets, C++, Matlab etc., final decision was to go with Clojure programming language (logo, fig. 14). Below, there are pro's and features of Clojure and why it is chosen to go along with:

- Functional programming, but there is sense of object orientation in the GUI part.
- Immutable data structures. Not very important for our project, but mainly supports concurrent programming and parallell processing thanks to this feature. Most of the variable types are immutable.
- It is inherited from both JAVA and LISP, as a result we have A.I. compatibleness and numerous dependencies/libraries.
- It has a handy REPL facility. One can even desing a game by simultaneous coding and getting the result.
- It is mainly used web applications (also see ClojureScript), games, A.I. and designs with concurrency.
- Although it is a rather new language, it is getting popular everyday.
- Hosted on JVM.
- Eagerness to learn a new language

To support the design with a nice interface, play-clj dependency is chosen, which is originally based on game design. Other options were Swing and SeeSaw. The play-clj ,in fact , is a wrapper for LibGDX of JAVA.

The best way to use Clojure is seems to be Leiningen, the project management tool. This great tool is itself coded in Clojure and allows you to create executables, compile, dependency check, create templates and so on. (logo, fig. 15)

Finally, one of the most Clojure-friendly editors is LightTable. It is also coded in Clojure and provide a flexible workspace. (logo, fig. 16)

Figure 14-16: Logos of development tools

A long part of the project was to create the working environment. Eventhough, project started on Windows 10, after encountering a problem, operating system is switched to Ubuntu 16.04. Following URLs are main websites for platforms, given in References. [8, 9, 10, 11]

4.2 Results

In literature, there was no general study on comparing coverage planning methods. Therefore, we came up with some parameters to present proper results. Namely, two variables are defined as follows:

$$Efficiency = 100\ x\ (\ 1 - \frac{T_{revisited}}{T_{total} - T_{obstacles}})$$

$$Difficulty = 10\ x\ \frac{T_{random\ obstacles}}{T_{total}}$$

Here, T letter stand for "number of ... tiles" notation. Moreover, $T_{random\ obstacles}$ and $T_{obstacles}$ may be different, as the former is parameter to be inserted before running to give an idea about difficulty. After, this value is used to randomize positions of obstacles and some of them may be the same. As a results, the latter parameter is acquired gives the number of obstacles with distinct coordinates.

Furthermore, there determined three different map sizes and 4 difficulty levels for each map size. To be more specific, we have small map (5x5 tiles), medium map (10x10) and large (20x20) map.

Below given the results of average efficiencies for randomized maps with distinc difficulty and map sizes in Table 1.

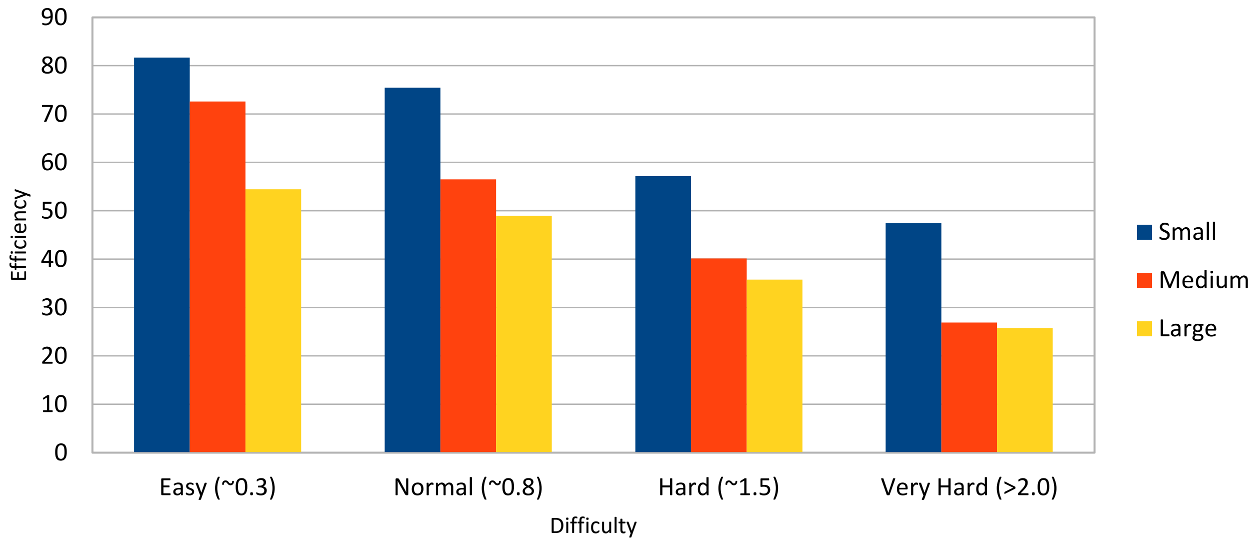

Table 1: Average efficiencies of distinct randomized maps by 4 difficulty levels

4.3 Discussion

As mentioned before, it is not possible to compare our method with others by mathematical means, but it is possible talk about its advantages and disadvantages.

Firstly, CleanBot is really efficient considering it is an online planner. With its 1-tile 4-connected proximity sensor, it is considered to be really blind for this task. But it can successfully fulfill the achievement in an efficient way. In fact, the numbers given in the Table 1 chart are really satisfying, because it is not possible to get an efficiency in most of the cases.

If there given a problem with a possibility of 100 % efficiency, CleanBot will either get hundred percent or go over 85 % anyway.

Secondly, the memory space to stack coordinates is rather small. There only visited coordinates, current left/right sweep lines and back-loop coordinates are stored. For example, in a large map, memory space of 450 x 2 = 900 numbers would be enough.

All in all, a simple planner is intentded that is capable of online searching with limited capacity. In the end, it seems to be successful after simulation results.

5. CONCLUSION

To conclude, as discussed in the former part, regarding simulation results and intuition of an online planner, CleanBot demonstrates an efficient way of covering the given map. This design, if transformed into real world hardware, seems to be able fulfill its job. Of course, in the real world applications, there would be much more other issues to be solved.

Implementing a simulation was a great work that tought Clojure, LISP and a little JAVA programming understanding. And it was helpful to show how to handle a planning/AI problem properly. This was a nice starting point to grasp the idea behind AI world.

After this part, in Future Work, some well-reasoned facilities are argued out in small detail. Then, references are given. Final part is Appendix for coding of whole project. This last part is included and commented so as to readers will have an idea on how Clojure looks like.

Please visit our website *volkanokbay.wixsite.com/cleanbot,* that will be updated until all the project material are completed to be published.

FUTURE WORK

In this part, some of future design additions are discussed. Note that, they still may be extendable and changeable.

A- **Battery Management :** As in all of the real world application, first upgrade should be addition of a battery station and charge consideration. That would be possible by calculation of charge needed to go back to station in every move, so that CleanBot would not be out of charge. Please notice that, in this situation, environment will be switched to “semidynamic”, as time goes into play. There can be more than one station, too.

B- **Moving Obstacles :** This is about considering a human or a pet that is sensed as an obstacle, but actually it is not. This problem can be achieved by machine learning and identifying living creatures.

C- **Concurrent Pollution :** In this case, environment will be totally dynamic.

D- **Doors and Rooms :** When it is possible to detect doors and it is required to finish a room before moving to a new one, tha planning should be altered.

E- **Multi Agent Operation :** This adds simply need of a communication network between bots.

F- **Hardware Implementation :** The ultimate goal would be realizing the simulation in a physical world. But it would be a whole new project, while there will be a lot more issues to be considered. Environment would probably change to a “stochastic” one.

REFERENCES

[1] Strimel, G.P (2014). "Map Learning and Coverage Planning for Robots in Large Unknown Environments ". Carnegie Mellon University, PA.
https://pdfs.semanticscholar.org/176a/580f6b01a036d7c1c508dfd2c6b934972827.pdf

[2] YouTube. "bObi Robotic Vacuum Cleaner and Mop"
https://www.youtube.com/watch?v=jrbseDjPVMU

"The Best Robot Vacuums 2016"

https://www.youtube.com/watch?v=9u-XwNcdKRU

"Smartest, Most Powerful, Best Robot Vacuum - 5 Vacuum Cleaners"

https://www.youtube.com/watch?v=Z3WCAbq8c4M

[3] LaValle, S.M. (2006). "Planning Algorithms". Cambridge University Press.

http://planning.cs.uiuc.edu/node1.html

[4] Yakoubi, M.A. & Laskri, M.T. (2016). "The path planning of cleaner robot for coverage region using Genetic Algorithms". Badji Mokhtar University, Algeria.

http://ac.els-cdn.com/S2352664516300050/1-s2.0-S2352664516300050-main.pdf?_tid=6f1c8a1a-cb99-11e6-853a-00000aacb360&acdnat=1482777233_dd1b9bcf949537d6c22901a7c190d6c7

[5] Choset, H. (2005). "Robotic Motion Planning: Cell Decompositions Lecture Notes". Carnegie Mellon University, PA.

(http://www.cs.cmu.edu/~motionplanning/lecture/Chap6-CellDecomp_howie.pdf

[6] Doty, K.L. & Harrison R.R. (1993). "SWEEP STRATEGIES FOR A SENSORY-DRIVEN, BEHAVIOR-BASED VACUUM CLEANING AGENT". University of Florida, FL.

https://www.aaai.org/Papers/Symposia/Fall/1993/FS-93-03/FS93-03-008.pdf

[7] Official website of the project CleanBot.

http://volkanokbay.wixsite.com/cleanbot

[8] Official website of the project CleanBot.

www.clojure.org

[9] Official website of the project CleanBot.

www.github.com/oakes/play-clj

[10] Official website of the project CleanBot.

www.leiningen.org

[11] Official website of the project CleanBot.

www.lighttable.com

[12] Wikipedia page for "Flood Fill Algorithm". https://en.wikipedia.org/wiki/Flood_fill

APPENDIX

A: Code

project.clj

```
;; -------------------------------------------------
;; Project             : CleanBot, vacuum cleaning robot
;; Project Type        : A simple simulation GUI for CleanBot's
motion planning
;; Project Designer : Volkan OKBAY 2016 (c) METU
;;
;; Written in Clojure 1.7, runs on JVM 6
;; -------------------------------------------------
```

Defines project and dependencies to be loaded.

```
(defproject cleanbot "0.0.1-SNAPSHOT"
  :description "FIXME: write description"

  :dependencies [[com.badlogicgames.gdx/gdx "1.9.3"]
                 [com.badlogicgames.gdx/gdx-backend-lwjgl "1.9.3"]
                 [com.badlogicgames.gdx/gdx-box2d "1.9.3"]
                 [com.badlogicgames.gdx/gdx-box2d-platform "1.9.3"
                  :classifier "natives-desktop"]
                 [com.badlogicgames.gdx/gdx-bullet "1.9.3"]
                 [com.badlogicgames.gdx/gdx-bullet-platform "1.9.3"
                  :classifier "natives-desktop"]
                 [com.badlogicgames.gdx/gdx-platform "1.9.3"
                  :classifier "natives-desktop"]
                 [org.clojure/clojure "1.7.0"]
                 [play-clj "1.1.1"]]
```

Some configurations for JAVA structure

```
  :source-paths ["src" "src-common"]
  :javac-options ["-target" "1.6" "-source" "1.6" "-Xlint:-options"]
  :aot [cleanbot.core.desktop-launcher]
  :main cleanbot.core.desktop-launcher)
```

desktop-launcher.clj

Defines namepace for desktop launcher.

```
(ns cleanbot.core.desktop-launcher
  (:require [cleanbot.core :refer :all])
  (:import [com.badlogic.gdx.backends.lwjgl LwjglApplication]
           [org.lwjgl.input Keyboard])
  (:gen-class))
```

Calls main function from core.clj file.

```
(defn -main
  []
  (LwjglApplication. cleanbot-game "cleanbot" 800 600)
  (Keyboard/enableRepeatEvents true))
```

core.clj

Namespace for core functions and required dependencies.

```
(ns cleanbot.core
  (:require [play-clj.core :refer :all]
            [play-clj.g2d :refer :all]
            [play-clj.ui :refer :all]
            [play-clj.math :refer :all]))

(declare cleanbot-game main-screen)
```

Generic parameters for different maps.

```
(def col# 22) ;; Number of columns
(def row# 22) ;; Number of rows
(def tile-size 50) ;; Pixel length of a single tile edge
(def time-quantum 0.06) ;; Time interval of a single move (in
seconds)
(def unvisited-val 0)
(def visited-val 2)
(def obs-val 1)
(def init-pos [1 1]) ;; Initial position
(def obs# 100) ;; Number of randomized obstacles
(def obs-list
  (partition 2
         (interleave
           (repeatedly obs#  #(inc (rand-int (- col# 2))))
(repeatedly obs#  #(inc (rand-int (- row# 2))))
         )
  )
)
```

```
;; Sample Map (col = 20 row = 10)
;; (def obs# 13)
;; (def obs-list [[6 4] [6 5] [5 4] [5 5] [11 5] [11 6]
;;              [12 3] [12 6] [13 3] [13 6] [14 3] [14 4] [14 5]])
```

Checks if all the map is covered.

```
(defn- map-covered? [map-vector]
  (not (some true? (map (partial some #(= unvisited-val %)) (map
vec map-vector))))
  )
```

Returns tile value.

```
(defn- tile-value [pos full-map direction]
  (let [tile-post (case direction
                 :up    [(first pos) (inc (second pos))]
                 :down  [(first pos) (dec (second pos))]
                 :left  [(dec (first pos)) (second pos)]
                 :right [(inc (first pos)) (second pos)])
     ]
    (aget full-map (- (dec row#) (second tile-post)) (first tile-post))
  )
)
```

Checks if given tile is an obstacle.

```
(defn- obstacle? [pos full-map direction]
  (if (= obs-val (tile-value pos full-map direction))
    true
    false
    )
  )
```

Checks if given tile is unvisited.

```
(defn- unvisited? [pos full-map direction]
  (if (= unvisited-val (tile-value pos full-map direction))
    true
    false
    )
  )
```

Checks if given tile is visited.

```
(defn- visited-tile? [map-vector pos direction]
  (if (< obs-val (tile-value pos map-vector direction)) ;; WARN
    true
    false
    )
  )
```

Checks if given tile exists in the sweep vector.

```
(defn- exists? [sweep-vector-list sweep-vector]
  (some #(= sweep-vector %) sweep-vector-list)
  )
```

Check state of sweeping. Are there tiles to be swept up/down remaining?

```
(defn- check-state [{:keys [robot? direction pos map-vector] :as
entity}]
  (if robot?
    (case direction
      :up (if (obstacle? pos map-vector :up)
            (assoc entity :unvisited-up-tiles-remain? false)
```

```
        entity)
    :down (if (obstacle? pos map-vector :down)
          (assoc entity :unvisited-down-tiles-remain? false)
          entity)
    :left (assoc entity :unvisited-down-tiles-remain? (unvisited?
pos map-vector :down)
                      :unvisited-up-tiles-remain? (unvisited? pos map-
vector :up))
    :right  (assoc entity :unvisited-down-tiles-remain?
(unvisited? pos map-vector :down)
                      :unvisited-up-tiles-remain? (unvisited? pos map-
vector :up))
   )
   entity
  )
)
```

Colors tile as white as it is vacuumed.

```
(defn- visit-tile [current_pos {:keys [robot? back? direction pos
map-vector cell-label] :as entity}]
   (cond
   robot? (assoc entity
         :map-vector (aset map-vector (- (dec row#) (second
pos)) (first pos) cell-label)
         )
   back?  (if (= pos current_pos)
         (texture! entity :set-texture (texture! (texture
"clean_tile_wood.jpg") :get-texture))
         )
   :else entity
   )
  entity
)
```

Updates counters for visited/revisited tiles.

```
(defn- visit-count [pos full-map {:keys [text? visit-counter revisit-
counter] :as entity}]
  (if text?
   (if (= visited-val (aget full-map (- (dec row#) (second pos))
(first pos)))
    (assoc entity :revisit-counter (inc revisit-counter))
    (if (= unvisited-val (aget full-map (- (dec row#) (second pos))
(first pos)))
     (assoc entity :visit-counter (inc visit-counter))
      entity)
   )
   entity
  )
)
```

Appends current tile to sweep lines.

```
(defn- get-sweep-vec [map-vector pos direction]
 [(tile-value pos map-vector direction) (case direction :right (inc
(first pos)) :left (dec (first pos))) (second pos)]
  )
```

Checks if there are tiles unvisited in given vector.

```
(defn- unvisited-left? [sweep-vector]
  (some #(= unvisited-val %) (map first sweep-vector))
  )
```

Get a move sequence from current position to closest new cell position by flood fill graph search through visited tiles.

```
(defn- go-to-cell [{:keys [pos map-vector back-loop-list] :as
entity} unvisited-cell-list]
 (let [map-vec (to-array-2d (map vec map-vector))]
   (def target-pos (drop 1 (last unvisited-cell-list)))
   (def step (atom 100))
   (aset map-vec (- (dec row#) (second pos)) (first pos) @step)
   (while (= (aget map-vec (- (dec row#) (second target-pos))
(first target-pos)) visited-val)
    (do
     (doseq [idx1 (range row#) idx2 (range col#)]
      (if (= (aget map-vec (- (dec row#) idx1) idx2) @step)
       (do
          (if (= (aget map-vec (- (dec row#) (inc idx1)) idx2) visited-
val) (aset map-vec (- (dec row#) (inc idx1)) idx2 (inc @step)))
          (if (= (aget map-vec (- (dec row#) (dec idx1)) idx2)
visited-val) (aset map-vec (- (dec row#) (dec idx1)) idx2 (inc
@step)))
          (if (= (aget map-vec (- (dec row#) idx1) (inc idx2)) visited-
val) (aset map-vec (- (dec row#) idx1) (inc idx2) (inc @step)))
          (if (= (aget map-vec (- (dec row#) idx1) (dec idx2))
visited-val) (aset map-vec (- (dec row#) idx1) (dec idx2) (inc
@step)))
          )
         )
        )
       (swap! step inc)
       )
      )
     (def xstep (atom (first target-pos)))
     (def ystep (atom (second target-pos)))
     (def move-list (atom []))
     (swap! step dec)
     (while (> @step 99)
      (do
       (cond
          (= (aget map-vec (- (dec row#) (inc @ystep)) @xstep)
@step) (do (swap! ystep inc) (swap! move-list (partial cons
:down)))
          (= (aget map-vec (- (dec row#) (dec @ystep)) @xstep)
@step) (do (swap! ystep dec) (swap! move-list (partial cons
:up)))
          (= (aget map-vec (- (dec row#) @ystep) (inc @xstep))
@step) (do (swap! xstep inc) (swap! move-list (partial cons
:left)))
          (= (aget map-vec (- (dec row#) @ystep) (dec @xstep))
@step) (do (swap! xstep dec) (swap! move-list (partial cons
:right)))
         )
        (swap! step dec)
       )
      )
    (assoc entity :sweep-mode :back-loop :back-loop-list @move-
list :unvisited-cells unvisited-cell-list)
   )
  )
```

Gets position for unvisited cells in sweep vectors.

```
(defn- get-unvisited-cell-pos [{:keys [pos map-vector sweep-left
sweep-right unvisited-cells] :as entity}]
  (apply
   (partial conj
         (apply (partial conj unvisited-cells)
              (map (fn [vec] [(first vec) (dec (second vec)) (last
vec)])
                   (filter #(= unvisited-val (first %)) (map first
(partition-by first sweep-right))))))
   (into [] (map (fn [vec] [(first vec) (inc (second vec)) (last vec)])
                    (filter #(= unvisited-val (first %)) (map first
(partition-by first sweep-left))))))
)
```

Clear visited cells from the back-loop list.

```
(defn- clear-unvisited-cell-list [full-map cell-list]
  (into []
   (cons [:bottom]
   (filterv
       #(or (unvisited? [(second %) (last %)] full-map :left)
(unvisited? [(second %) (last %)] full-map :right))
       (drop 1 cell-list)
   )))
)
```

Actual mode select function while cell decomposition.

```
(defn- cell-decomposition [{:keys [robot? sweep-right sweep-
left unvisited-cells] :as entity}]
  (if robot?
```

```
    (if (and (not (:unvisited-up-tiles-remain? entity)) (not
(:unvisited-down-tiles-remain? entity)))
     (if (unvisited-left? sweep-left)
      (assoc entity :sweep-mode :left :unvisited-cells (get-
unvisited-cell-pos entity))
      (if (unvisited-left? sweep-right)
       (assoc entity :sweep-mode :right :unvisited-cells (get-
unvisited-cell-pos entity))
       (if (= :bottom (last (flatten (clear-unvisited-cell-list (:map-
vector entity) unvisited-cells))))
        (assoc entity :sweep-mode :right)
        (go-to-cell entity (clear-unvisited-cell-list (:map-vector
entity) unvisited-cells)))))
     entity)
    entity)
  )
```

Moving robot to given position from current one.

```
(defn- move-robot [direction {:keys [robot? pos map-vector
sweep-right sweep-left] :as entity}]
 (if robot?
  (let [
    new-x  (case direction
           :right (+ (:x entity) tile-size)
           :left (- (:x entity) tile-size)
           :up (:x entity)
           :down (:x entity))
    new-y  (case direction
            :right (:y entity)
            :left (:y entity)
            :up (+ (:y entity) tile-size)
            :down (- (:y entity) tile-size))
    new-texture (case direction
             :right 270
             :left 90
             :up 0
             :down 180
             )
    new-pos (case direction
            :up    [(first pos) (inc (second pos))]
            :down  [(first pos) (dec (second pos))]
            :left  [(dec (first pos)) (second pos)]
            :right [(inc (first pos)) (second pos)])
    new-sweep-right (case direction
               :up (if (exists? sweep-right (get-sweep-vec map-
vector new-pos :right))
                    sweep-right
                    (conj sweep-right (get-sweep-vec map-
vector new-pos :right)))
               :down (if (exists? sweep-right (get-sweep-vec
map-vector new-pos :right))
                  sweep-right
                  (into [] (cons (get-sweep-vec map-vector
new-pos :right) sweep-right)))
               :right [(get-sweep-vec map-vector new-pos
:right)]
               :left [(get-sweep-vec map-vector new-pos
:right)])
    new-sweep-left (case direction
               :up (if (exists? sweep-left (get-sweep-vec map-
vector new-pos :left))
                    sweep-left
                     (conj sweep-left (get-sweep-vec map-
vector new-pos :left)))
               :down (if (exists? sweep-left (get-sweep-vec map-
vector new-pos :left))
                  sweep-left
                  (into [] (cons (get-sweep-vec map-vector
new-pos :left) sweep-left)))
               :right [(get-sweep-vec map-vector new-pos :left)]
               :left [(get-sweep-vec map-vector new-pos :left)])
   ]
   (assoc entity :x new-x :y new-y
    :pos new-pos :angle new-texture :direction direction
    :sweep-right new-sweep-right :sweep-left new-sweep-left)
  )
  entity)
 )
```

Most important function of planning motion and calling move function.

```
(defn- motion-planning [{:keys [robot? direction pos map-vector
back-loop-list] :as entity}]
 (if robot?
  (let [
    next-move  (if (or (= :left (:sweep-mode entity)) (= :right
(:sweep-mode entity)))
            (case direction
            :up (if (:unvisited-up-tiles-remain? entity)
               :up
               (if (:unvisited-down-tiles-remain? entity)
                :down
                (if (obstacle? pos map-vector (:sweep-mode
entity))
                 (if (obstacle? pos map-vector :up)
                  :down
                  :up)
                 (if (visited-tile? map-vector pos (:sweep-
mode entity))
                  (if (obstacle? pos map-vector :up)
                  :down
                  :up)
                  (:sweep-mode entity))
                 )
                )
               )
            :down (if (:unvisited-down-tiles-remain? entity)
                :down
                (if (:unvisited-up-tiles-remain? entity)
                 :up
                 (if (obstacle? pos map-vector (:sweep-mode
entity))
                  (if (obstacle? pos map-vector :down)
                   :up
                   :down)
                  (if (visited-tile? map-vector pos (:sweep-
mode entity))
                   (if (obstacle? pos map-vector :down)
                   :up
                   :down)
                   (:sweep-mode entity)))
                )
               )
            :left (if (obstacle? pos map-vector :up)
                (if (obstacle? pos map-vector :down)
                 (if (obstacle? pos map-vector :left)
                  :right
                  :left)
                 :down)
                :up)
            :right (if (obstacle? pos map-vector :up)
                (if (obstacle? pos map-vector :down)
                 (if (obstacle? pos map-vector :right)
                  :left
                  :right)
                 :down)
                :up)
            )

            (first back-loop-list)
           )
   ]
```

```
     (assoc (move-robot next-move entity) :back-loop-list (rest
back-loop-list))
    )
   entity
  )
 )
```

This special function defines screen behavior on first show, on each render, when resizing, on key press and on any timer ticks. All the GUI entities such as robot, tiles and counter texts are defined with initializations here.

```
(defscreen main-screen
 :on-show
 (fn [screen entities]
  (update! screen :renderer (stage) :camera (orthographic))
  (add-timer! screen :step-time time-quantum time-quantum)
  (add-timer! screen :visit-counter  0.005 time-quantum)
  (add-timer! screen :visit-time     0.01 time-quantum)
  (add-timer! screen :check-success  0.02 time-quantum)
  (add-timer! screen :check-time     0.03 time-quantum)
  (add-timer! screen :cell-decompose 0.04 time-quantum)
  (add-timer! screen :print-status   0.05 time-quantum)
;;    (add-timer! screen :flag-time   0.1 time-quantum)
  (let [full-map  (to-array-2d
             (vec (repeat row# (vec (repeat col# unvisited-val)))))
       dirty_tile (texture "dirty_tile_wood.jpg")
       obstacle_tile (texture "obstacle_tile_wood.jpg")
       clean_tile (texture "clean_tile_wood.jpg")
       robot (assoc (texture "robot.png")
             :x (* tile-size (first init-pos))
             :y (* tile-size (second init-pos))
             :pos init-pos
             :angle 0
             :height tile-size
             :width tile-size
             :direction :up
             :robot? true
             :unvisited-down-tiles-remain? false
             :unvisited-up-tiles-remain? false
             :sweep-right []
             :sweep-left []
             :cell-label 2
             :sweep-mode :right
             :unvisited-cells [[:bottom]]
             :back-loop-list []
             )
       background (for [row (range row#)]
              (for [col (range col#)]
              (assoc (texture dirty_tile)
                 :x (* col tile-size)
                 :y (* row tile-size)
                 :pos [col row]
                 :height tile-size
                 :width tile-size
                 :back? true)))
       text-entity (assoc (label "Revisited Tiles: /n Visited Tiles:"
(color :white))
                 :text? true
                 :x (/ tile-size 2)
                 :y (/ tile-size 4)
                 :revisit-counter 0
                 :visit-counter 0
                 :total-tiles (- (* (- row# 2) (- col# 2)) (count
(distinct obs-list))))]

    (doseq [idx1 (range row#) idx2 (range col#)]
       (if (or (= 0 idx1) (= 0 idx2) (= (dec row#) idx1) (= (dec col#)
idx2) (some #(when (= [idx2 idx1] %) %) obs-list))
         [(aset full-map (- (dec row#) idx1) idx2 obs-val)
          (texture! (nth (nth background idx1) idx2) :set-texture
(texture! obstacle_tile :get-texture))]
        )
    )
    (println "Initial Map:")
    (println (clojure.string/join "\n" (map vec full-map)))
    [ background
      text-entity
      (assoc robot
         :map-vector full-map
         :sweep-left [[(aget full-map (- (dec row#) (dec (first init-
pos))) (second init-pos)) (dec (first init-pos)) (second init-pos)]]
         :sweep-right [[(aget full-map (- (dec row#) (inc (first init-
pos))) (second init-pos)) (inc (first init-pos)) (second init-pos)]]
         :unvisited-down-tiles-remain? (unvisited? init-pos full-
map :down)
         :unvisited-up-tiles-remain? (unvisited? init-pos full-map
:up))
    ]
   )

  )

 :on-render
 (fn [screen entities]
  (clear!)
  (->> (for [entity entities]
      (if (:text? entity)
       (doto entity (label! :set-text (str "Revisited Tiles: "
(:revisit-counter entity)
                         "    Visited Tiles: " (:visit-counter
entity) "/" (:total-tiles entity)
                         "\nEfficiency:" (* 100 (- 1 (float (/
(:revisit-counter entity) (:total-tiles entity)))))
                         "    Difficulty:" (* 10 (float (/ obs# (*
(dec col#) (dec row#))))))))
       entity))
     (render! screen)))

 :on-resize
 (fn [screen entities]
  (if (>= row# col#)
   (height! screen (* tile-size row#))
  (width! screen (* tile-size col#)))
 )

 :on-key-down
 (fn [screen entities]
  (cond
   (key-pressed? :r) (app! :post-runnable #(set-screen!
cleanbot-game main-screen))
   :else entities))

 :on-timer
 (fn [screen entities]
  (case (:id screen)
   :step-time   [(map (partial motion-planning) entities)]
   :visit-counter [(map (partial visit-count (:pos (last entities))
(:map-vector (last entities))) entities)]
   :visit-time  [(map (partial visit-tile (:pos (last entities)))
entities)]
   :check-time  [(map (partial check-state) entities)]
   :cell-decompose [(map (partial cell-decomposition) entities)]
   :print-status [ (println "\nSTATUS REPORT")
          (println (clojure.string/join "\n" (map vec (:map-
vector (last entities)))))
          (println "Robot position:" (:pos (last entities)))
          (println "Unvisited down lines:" (:unvisited-down-
tiles-remain? (last entities)))
          (println "Unvisited up lines  :" (:unvisited-up-tiles-
remain? (last entities)))
          (println "Direction:" (:direction (last entities)))
          (println "Right sweep line:" (:sweep-right (last
entities)))
```

```
            (println "Left sweep line:" (:sweep-left (last
entities)))
            (println "Sweep mode:" (:sweep-mode (last
entities)))
            (println "Back loop move list:" (:back-loop-list (last
entities)))
            (println "Back loop cell list:" (distinct (:unvisited-
cells (last entities))))
            entities]
   :check-success (if (map-covered? (:map-vector (last
entities)))
             (do
              (do
               (remove-timer! screen :step-time )
               (remove-timer! screen :visit-counter)
               (remove-timer! screen :visit-time)
               (remove-timer! screen :check-time)
               (remove-timer! screen :print-status)
               (remove-timer! screen :cell-decompose)
               (remove-timer! screen :check-success)
               (println "All area is covered!")
               (println "Obstacle List:" obs-list)
              )
               entities)
             entities)
;;     :flag-time (println "FLAG")
   )
  )
 )
```

This header function defines the game and starts the main screen on the display.

```
(defgame cleanbot-game
 :on-create
 (fn [this]
  (set-screen! this main-screen)))
```