Media Engineering and Technology Faculty
German University in Cairo

Where are my Parts?

CSEN901 - Project 1

Mina Saad 19-0946
Ahmed ElSafty 19-2180
Khaled ElBhaey 19-5709

Contents

Chapter 1

Introduction

Where-are-my-parts (WAMP for short) is a simple intuitive game where the goal is to assemble broken robots in an efficient manner. The broken robot parts are scattered around the grid, where they can only move in 2D directions, till they hit an obstacle, another part, or the border of the grid and die. Thenceforth, search algorithms were used to efficiently assemble the Robot with least possible cost.

WAMP has few limitations, for instance, the generated grid may not be solvable, due to the fact that robot parts must travel in a specific direction till it hits something, or die (hits the grid wall). Furthermore, when two parts merge up, they occupy two grid blocks, thus bulk movements should be carefully managed. Finally, when a bulk of parts (two or more merged robot parts) move together, the cost increases, which had to be handled in the heuristic part.

Java language was used in implementing the search problem. The project is divided into 9 main classes covering our approach implementation, the program is written for an external agent who controls the robot parts, the agent is aware of all parts' locations, obstacles and limitations, which is the main reason search algorithms are desirable for this kind of problems.

Grid.java : responsible for creating a random grid/problem, carry out cloning jobs and apply changes to the grid.

SearchAlgorithm.java : implements the general search algorithm discussed in the lecture. **Heuristic.java** : calculates the heuristic value of a given node.

Part.java : ADT representing the robot part, including his location and neighbouring robot parts.

SearchProblem.java : an abstract class which represents the search problem discussed in the lecture.

1

SearchTreeNode.java : an abstract data type which represents the Search Tree node and structure discussed in the lecture.

Wamp.java : extends search algorithm class, it holds the algorithm implementation functions, the order of expansion queue and the main method.

WampOperator.java : represents the action/operator.

WampSearchProblem.java : extends Search Problem, checks for goal test and calculates path cost.

WampState.java : represents a state in the state space.

The approach followed, the Abstract Data Types used and heuristic implementation are discussed in more details next chapters.

Chapter 2

Implementation

2.1 ADT

The *SearchTreeNode* object can be represented as an ADT with six instance variables. The **state** this node corresponds to, the **parent node**, implemented by the same ADT, the **operator** applied to generate this node, the **depth** of the node in the tree, the **path cost** from the root and the **Heuristic** value of the node.

The *SearchProblem* ADT is characterized by the following variables. The **State Space** containing all states reachable by any sequence of actions, the **initial state**, and list of **operators** available for the agent. Moreover, the goal test is computed by the function **boolean goalTest(State state)**, that takes a state as a parameter and checks if all parts are assembled or not. Also, it calculates the **pathcost(Objects... operators)** which takes a list of operators along with the initial state in the ADT, such that the path cost is returned for every state. Moreover, the class has *transferFunction(State input, Operator operator)* that will generate the next possible state given any state and an operator.

Where-are-my-parts ADT is the main class, it extends search algorithm. It calls genGrid() to generate a grid with random width, height, number of robots and obstacles, the maximum size of the random grid is 8x8 for memory reasons, and minimum 3x3. After generating the grid, WAMP generate the list of operators and initialize the WAMP search problem. Moreover, Robot parts and obstacles count occupy 0%-25% of the area of the grid. After generating the grid, Wamp calls the search() method given the algorithm as an input to visualize a possible solution. Furthermore, the class holds all the algorithm implementation functions to be used whenever needed.

WAMP state ADT extends state class, which represents the state of our grid, the location of robot parts and obstacles, the number of connected parts and a hashmap of all next possible states, which can be used to calculate the path cost.

2.2 Main Functions

GetBulkRec()

GetBulkRec(Part p) is a function that returns all the parts attached to the current part p and all the parts attached to them recursively . For simplicity, it gets all parts attached in a bulk if part P is a subset of this bulk.

Transfer function()

transferFunction(State input, Operator operator) generate the next possible state given any state and an operator. In other words, its the function responsible for moving the parts across the grid, checking the validity of the moves.

ApplyAndClone()

ApplyAndClone(ArrayList¡Part¿ AdjacentParts, WampOperator Operator,int partX, int partY) is used to apply changes to the current state after cloning it. It creates a new grid, clone the obstacles and robot parts, and apply changes on all Adjacent Parts using the input Operator and the magnitude of the movement given(PartX,PartY). Furthermore, it gives the robot part some awareness of which part he merged with. Finally, it updates the grid by freeing the initial positions of the robot parts.

GenGrid()

GenGrid() generates a grid with random dimensions, robot parts' and obstacles locations.

Search()

Search(grid,strategy,visualise) manages the order of the expansion queue, checks the goal test, and call the various search algorithms implemented in WAMP.java. Finally, it prints and returns the solution and the steps to retrieve the final form, along with the path cost and number of expanded nodes.

Chapter 3

Algorithms

3.1   Implementation

The general idea is to have a nodes-to-get-expanded queue initialized at the beginning of every search approach. The technique of enqueuing the global queue differs from one search algorithm to another. while expanding, operators are applied to each and every part in the grid, which output a set of states to be enqueued.

A simple adjustment to the algorithm has been implemented, as the repeated identical nodes are removed from the queue, due to the fact that the children would be identical as well because every part moves with his adjacent parts, and that will create redundant sub-problems when other parts try the same operator.

Breadth First Search

BFS function is called whenever this queue is nonempty. The function dequeues the first node from the beginning of the queue, expand it, and adds the expanded nodes to the end of the queue, till the goal test is satisfied or there is no solution (empty queue).

Depth First Search

A to-be-expanded queue is initialized at the beginning of the search approach, and the DFS function is called whenever this queue is nonempty. The function dequeues the first node from the beginning of the queue, expand it, and adds the expanded nodes to the beginning of the queue, till the goal test is satisfied or there is no solution (empty queue).

Iterative Deepening Search

Iterative deepening simply sets a limit for the DFS function, thus, a limit is set at the beginning of the search, and the DFS function is called till the limit is reached, the goal test is satisfied or the queue get emptied.

Greedy Search

Greedy Search is set to expand the least heuristic valued node in the global queue of
expanded nodes. Thus, the search dequeues the first value in the queue, expand it, and
sort the children according to the heuristic value using insertion sort. It terminates if the
goal test is achieved or the queue get emptied. However, due to the nature of the Where-
are-my-parts problem, robots can keep hitting an obstacle forever. This incompleteness
gives the Greedy search an non-terminating manner in most cases.[1]

A* Search

The A* Search follows the same footsteps of the greedy. However, it takes to account
the number of steps moved so far from the root node up to the current node as a part of
the priority value in the queue. Thus, giving greedy search a completeness feature. i.e, a
node with a robot hitting the wall will increase his priority queue value, putting it a step
back for another node to try its luck.

---

[1]Due to the nature of the problem, All search algorithms have non-terminating property, because
when an object hits an obstacle using a specific operator, a new state is created with the same operator
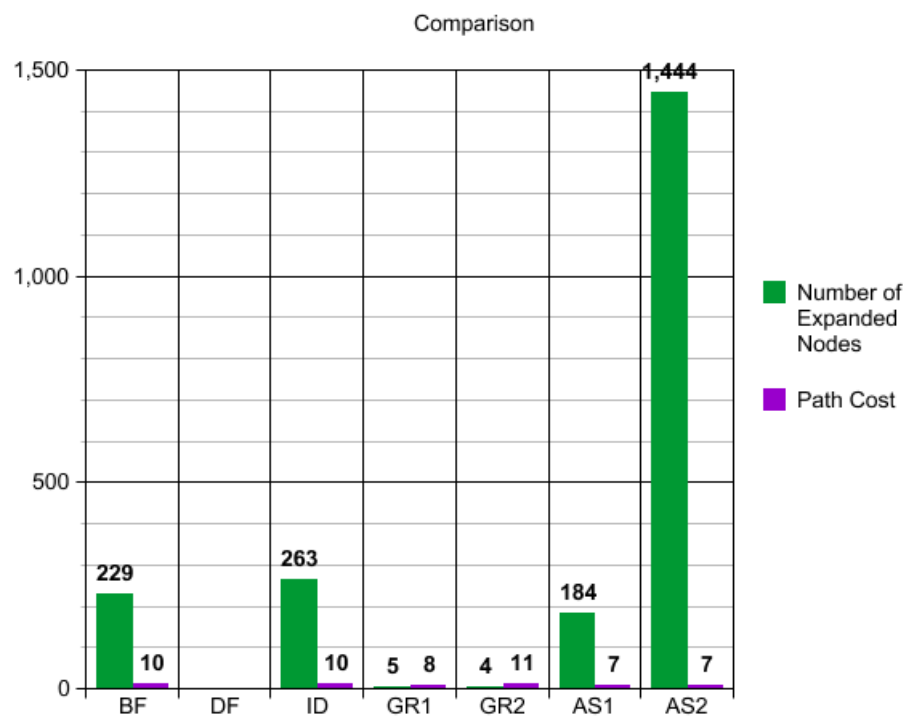instead of ignoring it

3.2 Comparison

Figure 3.1: Implemented Algorithms vs Number of expanded nodes. (1000 represent infinity
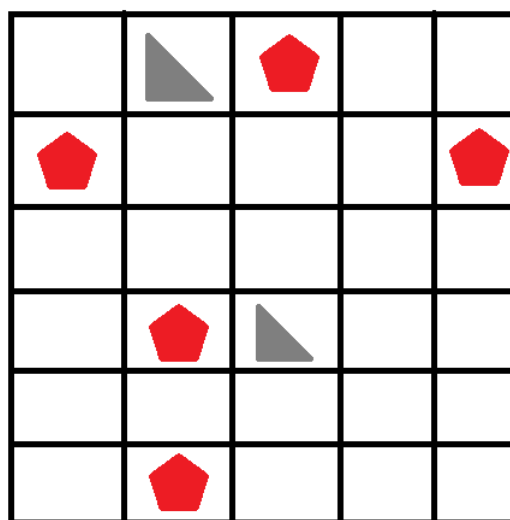

Figure 3.2: Test case used.

As shown in graph 3.1, Breadth-First search and Iterative deepening share same path cost. However, BF is more memory efficient as it expanded less number of nodes. The Greedy has the lowest number of expanded nodes due its features, with less path cost. However, Greedy algorithms are not complete in an unreliable way, such that, in most cases it tends not to terminate and find an answer. Finally, the A* search algorithm has the lowest optimum path cost from the initial state up to the goal state. However, the number of expanded nodes scales with the heuristic value and path cost, the lower the heuristic, the more nodes gets expanded.

Chapter 4

Heuristic

4.1 Centroid

The Centroid approach was used and implemented as a heuristic function in the project. As in, The heuristic function returns the average distance between the robot parts in the grid and the centroid point between them. As illustrated in figure 4.1, every robot parts' X and Y coordinates are modeled as a point in the grid. The points are used to create a virtual polygon by the aid of the geometry library "JTS". The Centroid is calculated using Polygon.getCentroid() function from JTS library. Finally, the average distance between all points and the centroid is returned as the heuristic value.
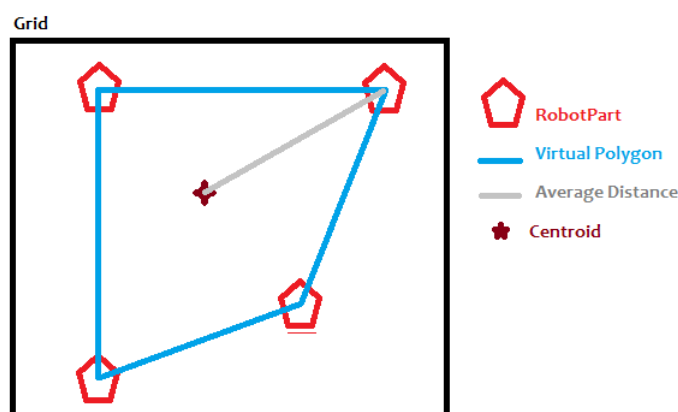

Figure 4.1: Centroid concept illustration

The heuristic approach proved to be a successful one, however it had a major drawback. If the Centroid can't be calculated because of the nature of the irregular shape e.g A twisted polygon as shown in figure 4.2. The heuristic will return infinity. which wasn't acceptable.
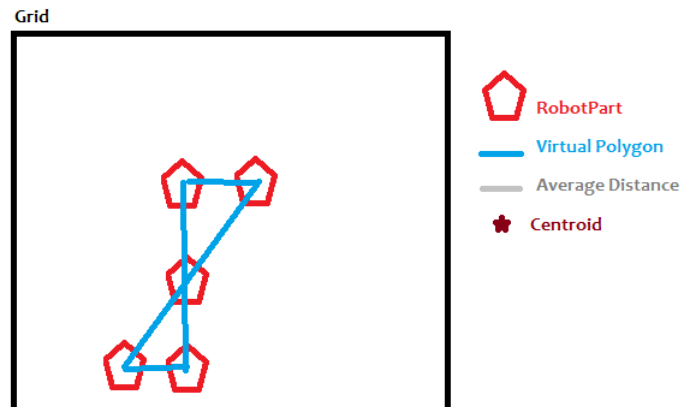
Figure 4.2: Centroid is incalculable incase of twisted polygons.

4.2   Average Distances

Due to the Centroid unsuccessful attempt, the average distances approach was used. As in, it gets the summation of all the distances between all parts and each others as shown in figure 4.3. The result is divided on the number of the parts Squared, to get an admissible average heuristic.
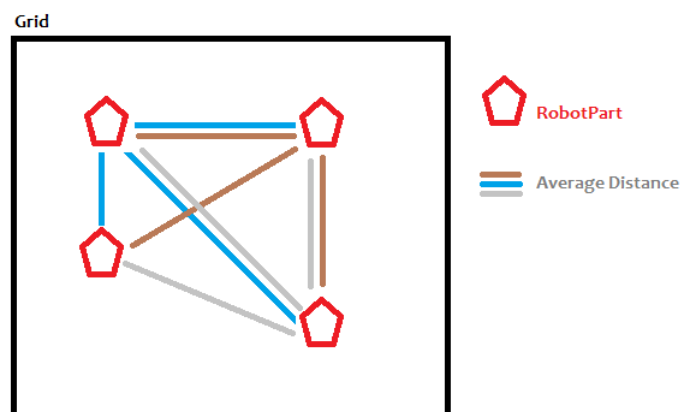

Figure 4.3: The distances between all parts and each other is used to evaluate the average distance.

The heuristic approach is satisfying the **Centering** property. The heuristic function will check for merged up parts, returning a zero when all parts are conjoined, and returning the average distance otherwise.

The approach followed is **monotonic (consistent)**. The heuristic value increases if one the parts stray away from the others, and decreases when it gains on other parts. The averaging is designed to get the Sum of distances and divide on the number of parts squared. Thus, avoiding any chance for any overestimation of heuristic. Moreover, parts move in 2D grid, the average distance between 2 parts in a grid is the hypotenuse in most cases, which is always shorter than the catheti. In case of straight lines, the distance is the twice the same as the heuristic value (assuming distance is in unit steps), which proves **admissibility**.

4.3   Number of Bulks remaining

The bumber of bulks remaining heuristic is a simple approach mainly used to assemble parts as fast as possible, The main concept is calculating the number of combined bulks of robot parts represented in the grid. Give more priority for small bulks to move, thus aiming to decrease the path cost by not moving the huge bulks. Also, it gives more priority for parts merging into other parts, rather than stopping at an obstacle.

The heuristic function is calculated as follows, the size of the bulks in the grid is doubled and added. Thus, bigger bulks will have larger output. The output is inversed to satisfy the **Monotonic** property by giving the bigger bulks less heuristic values. The function returns the number of parts in the board divided by the inversed value to satisfy the **admissibility** property, as the heuristic will always be less than one (the step cost). Finally, The **Centering** property is satisfied by a running check, returning a zero value if the number of bulks reaches one, i.e, there is only 1 merged robot.

Chapter 5

External Libraries

The JTS Topology Suite is an opensource API of 2D spatial predicates and functions. JTS geometry library was used in the project to implement the Centroid and average distances heuristics.

Centroid was calculated as follows, every robot part's coordinates creates a new JTS **coordinates(double x,double y)**, which its used as a parameter for a **linearRing(Coordinates[] s)**, that creates the virtual **polygon(LinearRing r)**, the Centroid can be easily calculated using **Polygon.getCentroid()** which returns the centroid as a Point.

Average Distances was calculated simply by using the following line, **DistanceOp. distance(Point x,Point y)**, where point takes coordinate as a parameter. Which returns the average distance between two given points.

Appendix

List of Figures