Student: Yahia Ahmed Saad El Gamal

ID: 19-1023

Tutorial: 14

Project: Where Are My Parts?

Discussion:

General Discussion:

This discussion will evolve around two parts, Environment Characteristics, and Analysis.

The problem of Where Are My Parts (WAMP) exhibits some characteristics. Here we will discuss the most important.
The only “thing” in the environment that is able to act is a robot part. Nothing else in the environment is changing the environment in a non-reflex way. A part can assemble when it hits another part. The search tree resulting from this problem has branching factor *b* proportional to the number of parts at the states. The greater the number of parts the greater the *b*. This makes the problem much much harder for greater number of parts.

Also in this problem, there are two important characteristics. The optimal solution is not necessarily in a lower depth than other non-optimal solution (which makes some algorithm like ID and BFS non-optimal). This is a consequence of another characters that the cost is not increasing as the depth increases.

The problem of as described in the project description exposes other challenges to any search algorithm aiming to solve it. One of the challenges is the huge number of possible cycles in the environment (any collision between a part and obstacle will lead to a cycle, not to mention many other scenarios where cycles can happen). The huge number of cycles will negatively effect many search algorithm and will almost make the DFS algorithm useless.

Implementation of the search-tree node:

```
class SearchNode(object):
    def __init__(self, state, parent_node, operator, depth, path_cost):
        raise NotImplementedError("Should have implemented this")

    def __str__(self):
        raise NotImplementedError("Should have implemented this”)
```

The SearchNode ADT is implemented as interface-like class in python. It only enforces any SearchNode to have two main methods, namely the constructer (`__init__`) and the `__str__` method.

The SearchNode follows the implementation of the text book in having a state, parent_node, operator, depth and path_cost.

Implementation of the search problem:

```
class SearchProblem(object):
    def __init__(self, initial_state):
        pass
    def operators(self):
        pass
   def goal_test(self, state):
        pass
    def path_cost(self, actions):
        pass
    def expand_node(self, node):
        pass
```

In the implementation of SearchProblem, the constructor took a mandatory initial_state argument. Other method interfaces followed the text book implementation in the sense that the state_space is an implicit function via a successor function and the operator functions.

In our actual implementation of the WAMP_SearchProblem, we have deviated a bit from the text-book implementation. As the set of operator were not a fixed set of operator as the set of operators change from one set to another. The successor function is also moved from the searchProblem to the WAMP_State (referred to as the Grid) in a an `apply_operator(self, operator)` method that applies an operator an returns a new State (a Grid).

Implementation of the WAMP problem ADT:

Of course the ADTs of general search were not sufficient to successfully implement and simulate the WAMP environment; a couple more of ADTs were added. Namely the Grid class and the Part class.

The Grid is where everything happens, it is the state (that is included in the SearchNode). The Part class represented an actual assembled parts and the assembly logic resides there.

The Grid interface is the following

```
class Grid(State):
    def __init__(self, grid=None):
    def possible_operators(self):
    def apply_operator(self, operator):
    @static_method
    def gen_grid()
```

Where the constructor uses `gen_grid` functionality for initializing the grid. possible_operators returns a set of possible operator (that is basically a cross product of parts and {N,E,S,W}). Other utility methods for assembly and motion are added but they are not a part of the public interface.

```
class WAMP_SearchNode(SearchNode):
    def __init__(self, state, parent_node=None,
                 operator=None, depth=0, path_cost=0):
    def expand(self):
```

This is the main functionality of a search node and this resembles the successor_function.

Main Function:

We have 4 modules, ADTs, WAMP, heuristics, search_queues. ADTs is where the abstract data type of search can be found. WAMP is an implementation of the interface in ADTs. heuristics is a module for heuristic function calculations, search_queues is different implementation of search queues to satisfy different queueing functions.

in WAMP:

```
def search(grid, strategy, visualize):
```

Is the main function for searching. It is behavior is as written in the project description (returning a string representing the path, cost of the path, number of nodes expanded).

There is also

```
def general_search(search_problem, nodes_q):
```

This function performs search on any given search_problem provided that the nodes_q (resembles the queuing function is provided).

in search_queues:
There are three classes of implemented queues. BFS_Queue, DFS_Queue, and BestFirst_Queue. The BFS is FiFo Queue, the DFS is LIFO, and the BestFirst is a priority queue. The implementation is standard with only a couple of notes in BestFirst_Queue

```
class BestFirst_Queue(SearchQueue):
    def __init__(self, cost_function, a_star=False):
```

where cost_function is a function_pointer to the heuristic function of concern. and a_star is a flag indicating whether the queue be prioritized by the path_cost + heuristic or just the heuristic value.

Search Algorithms:

All search algorithms implementation were reduced to choosing the right queue and using the general search except for iterative deepening algorithm that was implemented using a seprate search function (not the general_search method).

examples:
```
def dfs(search_problem, visualize=False):
    nodes_q = DFS_Queue()
    return general_search(search_problem, nodes_q)

def greedy(search_problem, heuristic_func, visualize=False):
    nodes_q = BestFirst_Queue(heuristic_func)
    return general_search(search_problem, nodes_q)

def A_star(search_problem, heuristic_func, visualize=False):
    nodes_q = BestFirst_Queue(heuristic_func, a_star=True)
    return general_search(search_problem, nodes_q)
```

Heuristic Functions:

Note¹: Finding a real “useful” admissible function in this problem is extremely hard. Of course one can make up any silly heuristic function (0 if goal, 1 otherwise), but this is not useful. Therefore, I will present a 4 heuristic functions, they all can sound *admissible* and I can try to make up arguments to support this claim (but I won’t). Anyhow, two of them is actually admissible (yet, it’s not very useful in some grids). Other heuristics can be found in the code under the module `heuristics`

Note²: Please note that in my environment, I assemble parts if they are in contagious squares. No need to bump parts together to assemble. This was written in the first section in the project description.

The first one is a very simple heuristic that is central but **NOT** admissible. It is simply

$$h(x) = num\_of\_parts(x) - 1$$

where num_of_parts(x) is the number of non-assembled parts in the grid x.

This is a much relaxed problem from the original problem, a problem in which each part can assemble with any part in only on step till there is only one big part (the actual robot). The “-1” part is responsible of the centrality characteristic, as the heuristic value will be zero if and only if there was only one part.

This heuristic is **consistent** (monotonic), as it will only increase if one part became two and this is not possible in our environment (proof by contradiction).

Sounds good? yet, this simple heuristic is **NOT** admissible, considering **Note²**, here is a simple counter example

R _ R
_ R _

$h(x) = 3 - 1 = 2$, yet the cost is equal to 1(as the part on the right or left will move towards the other parts making all parts assemble).

The Second one is not as simple as the first one, it is defined as

It simply returns $min[\ sum\_over\_p1(minimum\_dist(p1, p2) * num\_units(p2))\ ]\ For\ p1\ in\ parts, p2\ in\ parts, p1 \neq p2]$. In plain english, this heuristic tries to find a unit u such that the

```
def heuristic2(node):
    parts = node.state.parts
    min_dist = 99999999999
    for p1 in parts:
        dist = 0
        for p2 in parts:
            if p1 is p2:
                continue

            dist += minimum_dist(p1, p2) * len(p2.locations)

        if dist < min_dist:
            min_dist = dist

    return min_dist
```

distance travelled by all units to assemble to that *u* is minimized. And returns that minimized distance.

Why this is admissible **NOT**.
This is the actual cost of the best solution in a relaxed problem where assembled parts just stop wherever they want/need with no obstacles affecting the motion of the robots (or with the obstacles placed "just right"). Yet, this is not admissible because it didn't consider the change of distance needed to be travelled when for u3 when u2 assemble to u1. A simple counter example

```
_ R _
R _ R
_ X _
```

where the actual cost is 1, but the heuristic will produce 2. Even though this heuristic is not admissible but it

The third heuristic
This one is admissible,
This heuristic tries to assemble nearest parts and considers best case scenario in which all parts decrease the distance between the next part and the original part. (code is better than english)

```
def admissible1(node):
    grid = node.state
    units = map(lambda x: x.locations, grid.parts)
    units = [tuple(u) for l in units for u in l]
    min_sum = 999999999999
    i = -1
    for unit1 in units:
        i += 1
        current_parts_length = 0
        current_dist = 0
        for unit2 in sorted(units, key=lambda x: dist(unit1, x)):
            current_dist += max([0, dist(unit1, unit2) - current_parts_length])
            current_parts_length += 1

        if current_dist < min_sum:
            min_sum = current_dist

   if len(grid.parts) > 1:
        min_sum = max([1, min_sum])

    return min_sum
```

Why this heuristic is admissible.
The problem with the last heuristic was that when a unit u1 assembles to another unit u2 (having a part p1 consisting of 2 units), the job of assembling another unit u3 *might* be easier if u2 assembled with u1 in a way such that the minimum distance between u3 and p1 is less than the distance between u3 and u1. This heuristic calculates the best case scenario. Meaning that it assembles nearest parts, and assumes that those assembly will help new parts gets assembled ensuring a best-case cost of the relaxed problem mentioned in heuristic (and therefore, never over-estimates).

The fourth heuristic is admissible. It is calculated as

$min(max\_dist\_to\_parts(pi, P)\ \ \forall\ pi \in P)$

where max_dist_parts(part, part_list) calculates the maximum distance between *part* and *pj* for $pj \in$ *part_list*. The distance between parts is calculated by
part_dist(p1, p2) = min(cartesian_dist(ui, uj) $\forall$ *ui* $\in$ *p1* $\forall$*uj* $\in$ *p2))*. Where cartesian distance is calculated normally *(abs(x1 - x2) + abs(y1 - y2))*

```
def minimum_dist(part1, part2):
    return min([dist(a, b)
                for a in part1.locations
                for b in part2.locations])


def max_dist_to_parts(part, parts):
    if len(parts) == 1:
        return 0
    return max([minimum_dist(part, part2) - 1 for part2 in parts
                if part2 != part])

# min(max_dist_to_parts(pi, P) for all ui in P)
def admissible2(node):
    grid = node.state
    parts = grid.parts
    return min([max_dist_to_parts(p, parts) for p in parts])
```

This heuristic is better than the **third** one, it considers the change in distance needed for later parts when earlier parts assemble. Yet it also dominates the **third** heuristic (which is admissible).
So how and why this heuristic work?
The argument behind this heuristic is as follows,
To reach a goal state, all parts must assemble to one part,
In a relaxed problem in which obstacles don't exist and parts can stop whenever they want, the locations of the robot (after assembly of all units/parts) will reside in a set of locations L such that

$\exists\, p \in P \wedge places(pi) \subset L$
where P is the set of all parts, *places(part)* is a set of locations for *part* and L is the set of locations of the robot in the goal state
This means that at least one part didn't need to move.
This also means that for this part *p,* all parts needed to assemble with it (or assemble with other parts that assembled earlier with *p.* Otherwise it won't be the best-case scenario in the relaxed version).
Given we know that part *p* is not assembled with any other part,
And there is at least a movement of cost 1[1] for two parts to be assembled,
Then the cost of having all parts to assemble to *p* (or to assemble with assembled parts of p) <u>will not exceed the distance between part *p* and the furtherest part.</u> This is true because it considers the probable change of *distance* between other parts and *p* as more parts get assembled to *p. How ?*
The distance needed for part *p_last* (the last part to assemble to *p*) to assemble with part *p* might decrease when an intermediate part *p_intermediate* assemble with *p*. If part *p_intermediate* assembles with *p* and this assembly decreases the distance between *p_last* and *p*, the cost of this movement (moving p_intermediate to assemble with *p*) is at least equal to the decrease in the distance).
If there is no *p_intermdiate*, then the cost in the best case would be the actual distance between *p_last* and and *p*.

[1] More precisely, it is min(number_of_units(p1), number_of_units(p2))

So to assemble an intermediate part such that the distance between *p_last* and *p is decreased by d*, the cost of this assembly will be ≥ *d*. Therefore, the cost of assembly of all parts in the optimal solution will be ≤ max_dist_to(*p*). Therefore, we try to find *p* such that his value is minimized and it will be the cost of the best case scenario in the relaxed version.

Performance Measures:

The performance of a search algorithm varies as the grid varies. A high-level analysis will show us that (which differs from the standard text-book analysis as our problem doesn't have the non-decreasing cost characteristic).

Note: For A* and greedy to work given that the cost of bumping to an obstacle without moving is 0, we needed to make a small repeated states detection. Namley by checking only the parent node. such that if there are other repeated states, they won't have the same cost. (any operator of cost > 0, changes the state).

Table 1

Algorithm\Feature	Complete	Optimal
BFS	Yes	No
DFS	No	No
ID	Yes	No
Greedy	No	No
A*	Yes	Yes

The reason why BFS and ID aren't optimal is the non-strictly-increasing characteristic of the edge costs. Meaning that an edge in depth 1 might have higher cost than an edge in depth 20.

Numbers wise, there is no definitive answer. Due to the characteristics of greedy, sometimes greedy “embarrasses” other algorithms.

For the sake of this performance measure, it would be unfair to compare algorithms that are complete or optimal with algorithms that aren't. Therefore, we will emphasize on algorithms that have the same features as shown in Table 1. Namely (BFS, ID) (DFS, Greedy) and A* will not be comparable with any. **Yet** we will use the same grid and generate numbers to put everything in context.

Example 1:

```
_ _ _ _ _ _ _ _
X _ _ _ _ 2 2 _
X _ _ 3 3 _ _ _
_ _ X _ _ _ _ _
0 _ _ _ _ 5 X _
_ _ _ _ _ 5 5 _
_ 4 _ 1 _ 5 _ _
_ 4 _ _ _ _ _ _
```

	# expanded nodes	# max queue length	solution cost
BFS	3326	28764	29
DFS	NA	NA	NA
ID	181	32	29
GR1	4	28	20
GR2	4	29	26
A*1	3248	10278	15
A*2	1588	5440	15

Example 2:

```
X _ _ 0 _ X
_ _ _ _ _ _
_ _ _ _ 1 X
_ _ _ _ _ _
_ 2 _ _ 3 _
_ _ 4 _ _ _
5 X _ _ _ 6
```

	# expanded nodes	**# max queue length**	**solution cost**
BFS	35031	79593	25
DFS	NA	NA	NA
ID	4425	26	30
GR1	7	23	26
GR2	100	34	45
A*1	118241	164916	21
A*2	57633	94988	21

The difference in performance between A*1 and A*2 is expected as we already know that `admissible2` dominates admissible1. Also we we found that BFS reached a solution in a much smaller number of expanded of nodes, but the solution found wasn't optimal as it resided in a shallower depth (that's why it didn't expand/enqueue as much as the A*)