Problem of Shared Bike Management

Qiwei Qiu, *Student, ShanghaiTech,* Shengjia Zhang, *Student, ShanghaiTech,* Fang Chen, *Student, ShanghaiTech,* Sihan Zhang, *Student, ShanghaiTech,*

***Abstract*—This paper solve a shared bike management problem with translating it to a graph and do graph search on it. The UCS strategy have stationary performance, where A-star algorithm have better performance with suitable heuristic function.**

***Index Terms*—Shared bike, Graph search, A-star algorithm**

I. Introduction

In our daily lives, shared bikes appear in many places: neighborhoods, companies, metro station. When we want to use bikes, extreme circumstances exist that there are almost no bikes in neighborhoods and companies in the morning during the rush hours, while almost all the bikes are at the metro station. However, most of the potential riders starting point is his or her housing estate. The unreasonable distribution of the bikes is not only a waste of resources, but it is also inconvenient for almost everyone.

II. Model

Assume that there are many bike stations in a city, each stations have current bike number more or less than ideal number, we need to use a big truck, drive to these stations, get bikes onto the truck or put bikes off the truck, to meet requirement of each stations. Finally, all stations have bike number equal to ideal number. At the same time, we pay oil cost when drive the truck from one station to another, so we want to pay lowest cost on the road to achieve our goal.

We build model as graph, where nodes represent bike stations, and edges represent roads between stations, weight of edges represent cost of pass the road. Then we denote requirement of bike station as status of node. For example, if a bike station have n bikes more, we said $node.status = +n$, and if a bike station have n bikes less, we said $node.status = -n$.

Now the problem become to found a path through all unmanaged station and meet their requirement with lowest total cost. But this shortest path is hard to find. So we need to translate the graph to become easy to search. Also, to make problem simpler, we assume that when our truck arrive a station, we can only meet its requirement exactly or do nothing, not partly. For example, if a station need 5 bikes, we can only give 5 bikes to it, not partly give 2 or 3 bikes to it.

We build a new graph and search on this graph. In new graph, each node represent a state on old graph, that our truck stop in some place and every station need increase or decrease bike number. Define one action as our truck drive to a station through shortest path and meet requirement of this station. In new graph, nodes are connected with edge only if one node can be reached from another node through one action, and weight of edge is cost of that action. It is obvious that new graph is a directed acyclic graph. Our start state is a node that truck at start place and each station have initial requirement. Our end state is a node that truck at end place and each station have bike number equal to ideal number. Our goal is to search for a shortest path from start state to end state.

III. Algorithm

Algorithm 1 is a function $Search$ which receive an input of start state, including all stations, requirement of stations, path between stations and truck start position, then search for the shortest path to meet all requirement.

Algorithm 1 Search for shortest path

```
Input: Start state $s0$
Output: Shortest path $p$
1:  function SEARCH($s0$)
2:      Initialize state $s = s0$
3:      Initialize list $fringe = empty\_list$
4:      Initialize list $path = empty\_list$
5:      while $s$ is not end state do
6:          $fringe.insert(Successor(s))$
7:          $s = Strategy(fringe)$
8:      end while
9:      while $s \neq s0$ do
10:         $path.insert\_front(s)$
11:         $s = s.parent$
12:     end while
13:     $path.insert\_front(s0)$
14:     return $path$
15: end function
```

Algorithm 2 is a function $Successor$ which receive an input of state then give a list of successors of this state. Each successor represent that our truck drive to a station then meet its requirement, which means the station should be satisfiable. If our truck have n bikes now and capacity is m, then the arrived station should have $-n \leq node.status \leq m - n$.

Algorithm 3 is many kind of $Strategy$ which receive an input of a list of reached states and give the state to be opened next time. Depth first search (DFS) and breadth first search (BFS) are simple strategy but cost much to found shortest path. Greedy algorithm can found a solution within small time but probably not optimal. Uniform cost search (UCS) is good to found optimal solution in small time, where A-star algorithm have better performance with a good heuristic function.

IV. Conclusion

Good heuristic function depend on real data, so to design a good heuristic function to have better performance, we need to

Algorithm 2 Successors of state

```
Input: State $s0$
Output: Successors $successor\_list$
function SUCCESSOR($s0$)
    Initialize list $successor\_list = empty\_list$
    Initialize state $s$
    for $i$ in $s0.nodes().len()$ do
        if $s0.node(i).status \neq 0$ then
            $min = -s0.current\_load()$
            $max = s0.capacity() - s0.current\_load()$
            if $min \leq s0.nodes(i).status \leq max$ then
                $s = s0.copy()$
                $s.parent = s0$
                $cost = Distance(s0.current\_position, s0.nodes(i))$
                $s.cost = s0.cost + cost$
                $s.nodes(i).status = 0$
                $s.current\_load = s.current\_load + s0.nodes(i).status$
                $s.current\_position = s.nodes(i)$
                $successor\_list.insert(s)$
            end if
        end if
    end for
    return $successor\_list$
end function
```

have a broad vision of real city shared bike resource situation, which is a deep problem and could do many survey on it.

Algorithm 3 Strategy

```
Input: List of state $fringe$
Output: Next opened state $s$
function DFS($fringe$)
    $s = fringe.tail()$
    return $s$
end function
function BFS($fringe$)
    $s = fringe.head()$
    return $s$
end function
function GREEDY($fringe$)
    Initialize state $s\_best, s\_best.evaluate() = -\infty$
    for $s$ in $fringe$ do
        if $s.evaluate() > s\_best.evaluate()$ then
            $s\_best = s$
        end if
    end for
    return $s\_best$
end function
function UCS($fringe$)
    Initialize state $s\_min\_cost, s\_min\_cost.cost = +\infty$
    for $s$ in $fringe$ do
        if $s.cost < s\_min\_cost.cost$ then
            $s\_min\_cost = s$
        end if
    end for
    return $s\_min\_cost$
end function
function A-STAR($fringe$)
    Initialize state $s\_best$
    $s\_best.cost = +\infty, s\_best.h() = +\infty$
    for $s$ in $fringe$ do
        if $s.cost + s.h() < s\_best.cost + s\_best.h()$ then
            $s\_best = s$
        end if
    end for
    return $s\_best$
end function
```