Bicing

Tomas Barton and ***Mauro Donadeo***

November 3, 2009

1 Introduction

The aim of this project is to provide an program which would help the provider of local bicing service[1] with distribution of bikes to approach to an ideal state, when each user of this service would find a bike when he needs it.

1.1 About Bicing

The Bicing project is quite new in Barcelona (started in 2007[2]), but after few months it became very popular. Owner of a special card, which can be purchased for 30 € per year, can rent a bike from a station for free, if he returns this bike to some station in a half an hour. Otherwise he pays a few cents for each hour until the bike is returned.

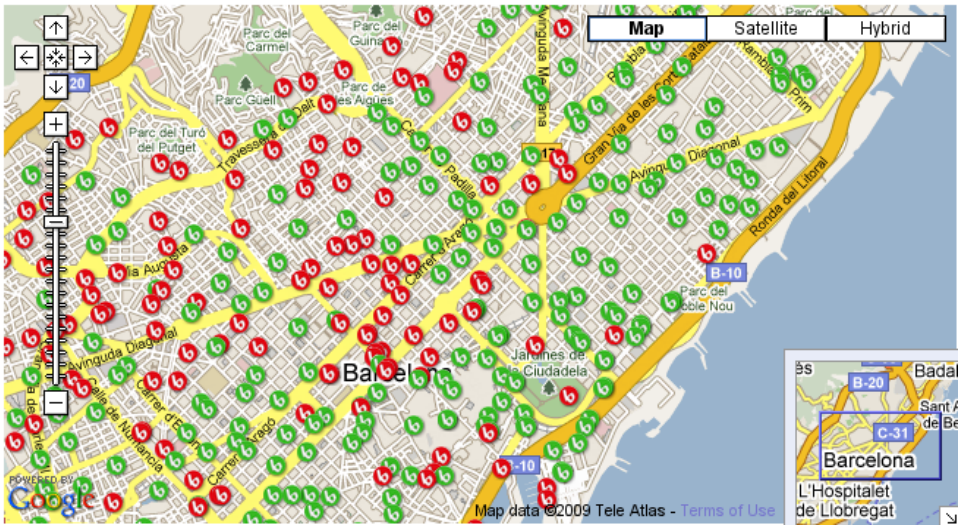

Image 1.1: Map of bicing stations in Barcelona – red stations are without bikes

Nowadays there are more than 400[2] bike stations in Barcelona, the highest concentration of bike stations is of course in centre of city. Usually each station has from 20 up to 30 stands for bikes. In the very center they are placed quite close to each other.

[1] Bicing Barcelona – `http://www.bicing.com`
[2] 418 stations at 12 October 2009

1.2 Our task

With usage of $\boldsymbol{F}$ vans we are supposed to optimize distribution of bikes, so that as many customers as possible will find a available bike at time when they need it.

A van can load up to 30 bikes and unload them to one or two another stations. The operator will supply us usage of bikes from which we can optimize moves of bikes.

Implementation has been done in Java, we took advantage of AIMA framework[1], which contains most of algorithms commonly used for searching state space.

1.3 Complexity of problem

As long as we have to search just for solution of this problem for upcoming hour, size of state space is significantly reduced. When we display all possible states of this problem as a n-ary tree, we get not really deep tree – this is exactly the number of maximum moves we can do and that is $\boldsymbol{F}$, but this tree can have a huge branching factor.

Number of different moves between stations can be approximately computed as:

$$moves = F * 2 * \begin{pmatrix} n \\ 2 \end{pmatrix} \tag{1.1}$$

where n – is number of stations
F – number of vans

We are looking for different pairs of stations and we can move bikes in both directions, so that we multiply combination by 2. In this formula we do not consider moves of different number of bikes between same stations, we also neglect moves of bikes to another two stations.

$$\begin{aligned} moves &= F * 2 * \left(\frac{n!}{2 * (n-2)!} \right) \\ &= F * 2 * \frac{n * (n-1)(n-2)!}{2 * (n-2)!} \\ &= F * n * (n-1) \end{aligned}$$

So that polynomial complexity of this problem seems to be:

$$T(n) = O(k * n^2) = O(n^2) \tag{1.2}$$

2 Implementation

2.1 State representation

Each state must contain information about distribution of bikes over all stations. So that we have to keep:

- current number of bikes at the station
- number of bikes that is going to be at station in an hour
- moves how to reach this state (from empty state)

These information are different for each state, moreover we have other information which we need but they are same for every state:

- number of stations
- coordinates of stations
- expected demand for bikes in upcoming hour

2.2 Operators

For the sake of creating new state we have implemented a set of operators. Firstly we have operator `addMove` for moving bikes from one station to another. It is quite clear that this operator can be used only when we have some not used bikes at a station. A bit more complicated case is when we load bikes at one station and unload them at two other stations, for this purpose we have `doubleMove` operator.

Secondly there is operator for altering already done move, which is called `changeMove` and we allowed changing number of bikes which are going to move and the destination. This operator checks availability of requested bikes at the station, in case that the number of bikes is not at the station, operator returns `false`. There is also possibility of changing source station but there might not be any bikes available and furthermore this action can be done by combination of `removeMove` and `addMove` (with requested source station).

Lastly we have already mentioned operator `removeMove` which delete move and return bikes to original location.

Here is a sort overview of all operators:

addMove simply moves given number of bikes from one station to another

doubleMove loads $a + b$ bikes at one station and unload a bikes at first station and b bikes at second station

changeMove alter number of bikes or destination of this move

removeMove deletes specified move

Station #	Not used	Next	Demand
1	10	10	0
2	0	0	0
3	0	0	2
4	0	0	8

Table 2.1: Initial state with heristic value $h_1 = 10.0$

Station #	Not used	Next	Demand
1	0	10	0
2	0	6	0
3	0	0	2
4	0	4	8

Table 2.2: Initial state with heristic value $h_1 = 6.0$

2.3 Heuristic functions

Heuristic function is supposed to evaluate state, so that searching algorithm can unambigously determine which state is better.

Version 1

Firstly, we have to take in consideration number of unsatisfied demand of bikes. If there are no bikes needed, or the demand is satisfied, the value of heuristic function would be 0. We are trying to minimize this formula:

$$h_1 = \sum_{i=0}^{n} not\_satistified\_demand_i \tag{2.1}$$

Let us take into consideration situation when we have 4 stations and bike are all bikes are located only on first station, as it is described in table 2.3.

The output of this function seems to be correct, if we move 4 bikes to station number 4, we would lower heuristic value to 6, but if we made nonsense move with other 6 bikes to station 2 (as is shown in 2.3), the value of heuristic function would be the same, so that we introduced version 2.

Version 2

This heuristic function proceed from version 1 however it takes into account also bikes which are not going to be used in next hour. Like in previous

version we are trying to minimize *h_2* value.

$$h_2 = \sum_{i=0}^{n} not\_satistifed\_demand_i + not\_used\_bikes\_in\_next\_hour_i \quad (2.2)$$

This function would penalize previously shown example but in some cases it would not be possible to reach score 0.0 for best solution.

Counting with distance

Next generation of heuristic function should also optimize solution according to total distace which would vans run while moving bikes. We proceed from heuristic function 2, which seems to give better results.

$$h_3 = h_2 + rate * \sum_{i=0}^{n} distance_i \quad (2.3)$$

Because currently used heuristic value and sum of distances use completly different units value of *rate* is completly experimental value.

2.4 Successor function

Successor function is called by searching algorithm to generate next states for expansion. This function does not evaluation of current state, this is done by heuristic function.

Version 1

First version of successor function generates for each station, where are available bikes, all possible moves of bikes to every other stations and the same with double moves. This state space can be displayed as a n-ary tree.

Version 2

Second version is made for simulated annealing, in each state we also try to alter or delete already done moves, which makes accessible one state from other states, so our state space is no longer a tree but a common graph.

3 Experiments

3.1 Simulated Annealing

Result of *hill climbing* function seems to be pretty good, by modifing parameters of *simulated annealing* algorithm we will try to get equivalent or better results like we had from *hill climbing* algorithm.

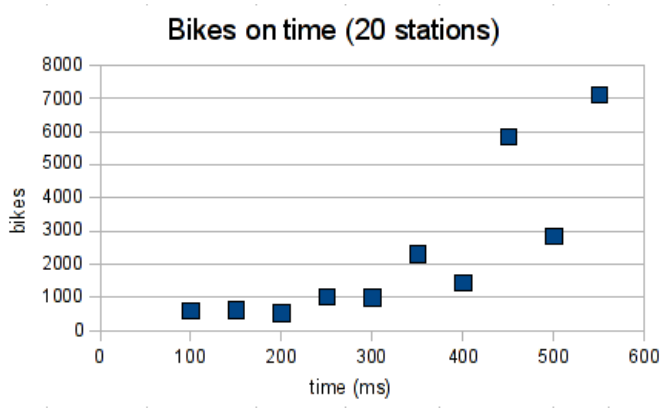

Image 3.1: Dependency of number of bikes on solution time

Parameters: *stations* = 20
mode of demand = RUSH HOUR
vans = 2
heuristic = 2
$k = 60$
$\lambda = 0.2$
limit = 100

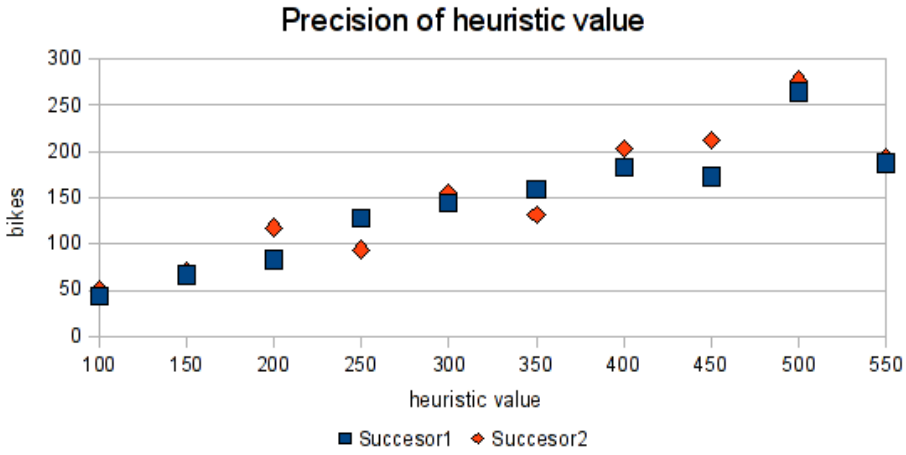

Image 3.2: Dependency of different successor functions on heuristic value

As you can see from 3.4

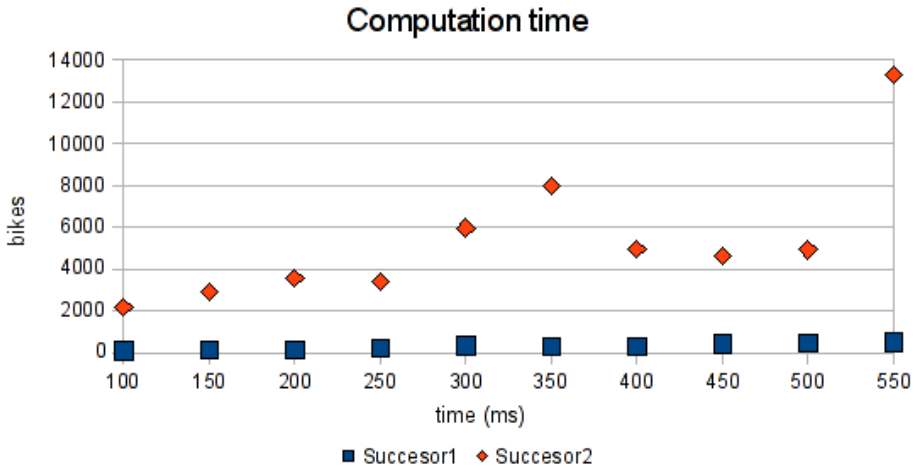

Image 3.3: Dependency of different successor functions on computation time

4 Conclusion

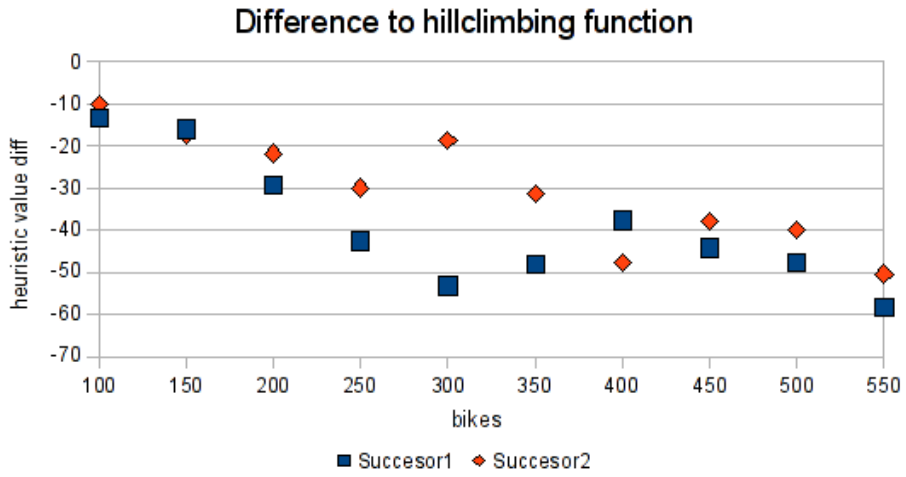

Image 3.4: Difference in heuristic value to Hill Climbing (the less the better))

5 Bibliography

[1] *Stuart Russell and Peter Norvig:***Artificial Intelligence: A Modern Approach** `http://aima.cs.berkeley.edu/`
Downloaded java implementation of algorithms (AIMA framework) from repository `http://code.google.com/p/aima-java/` by Ravi Mohan.
Date of publication: October 3, 2009.
Date retrieved: October 12, 2009.

[2] *Wikipedia:* **Bicing** `http://en.wikipedia.org/wiki/Bicing`
Date visited: October 27, 2009