Intelligent Systems

- Milestone 3 -

Juan Garrido Arcos
juan.garrido3@alu.uclm.es

Pedro-Manuel Gómez-Portillo López
pedromanuel.gomezportillo@alu.uclm.es

Abstract

In this milestone we had been asked for developing a basic versions of a search algorithm following 3 different strategies; breath-first search, depth-first search (depth-limited search and iterative deeping search) and uniform cost.

We are programming in python 2.7, using emacs24 as IDE and bitbucket as code control version. The result of the comparative between the studied data structures have been obtained under an Intel CoreDuo Ubuntu 14.04 with 3,8 GB of RAM.

Comparative between data structures

We were asked to choose an appropriate data structure for storing the fringe of our problem. For so, we studied two different alternatives,

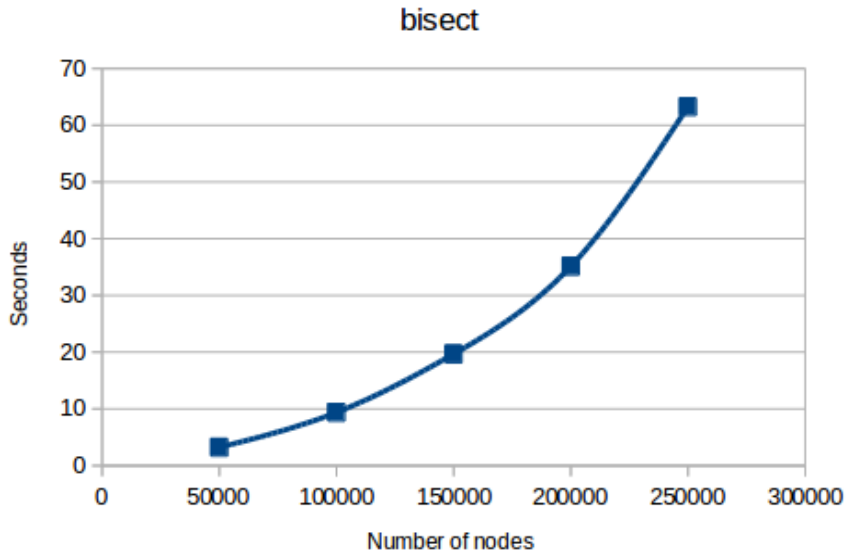
- basic *pythonic* list managed by **bisect**, a library named so because it uses a basic bisection algorithm for orderly inserting elements into a list
  API - https://docs.python.org/2/library/bisect.html

- **heapq**, a library which implements a priority queue by means of a binary tree
  API - https://docs.python.org/2/library/heapq.html

By using **bisect** for managing our fringe, we obtained the following results after one minute of execution,

Number of nodes in the fringe	Time to accomplish so
50,000	3.218
100,000	9.379
150,000	19.672
200,000	35.112
250,000	63.254

By using the GNU/Linux tool *System Monitor* we were able to compute how many MB of RAM the program did generate, and it was not even a 15% of our PC capacity, just around 400MB.



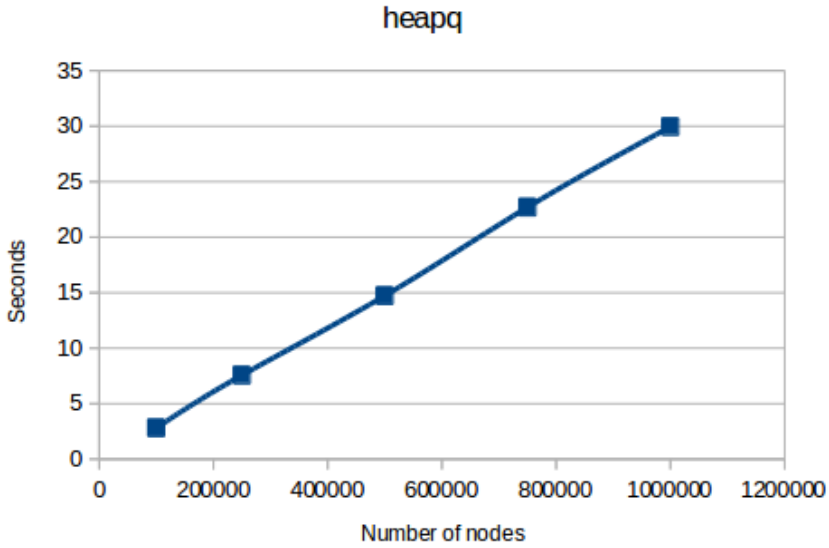
It can be seen that the plot of this function is following an exponential growth.

After studying **heapq** we collected results shown bellow,

Number of nodes in the fringe	Time to accomplish so
100,000	2.808
250,000	7.561
500,000	14.723
750,000	22.696
1,000,000	29.949

Passed 30 seconds we had to kill the process as the computer in which we were testing it ran out of RAM; a total of 70% of 3,8 GB of RAM were occupied just before killing the process.

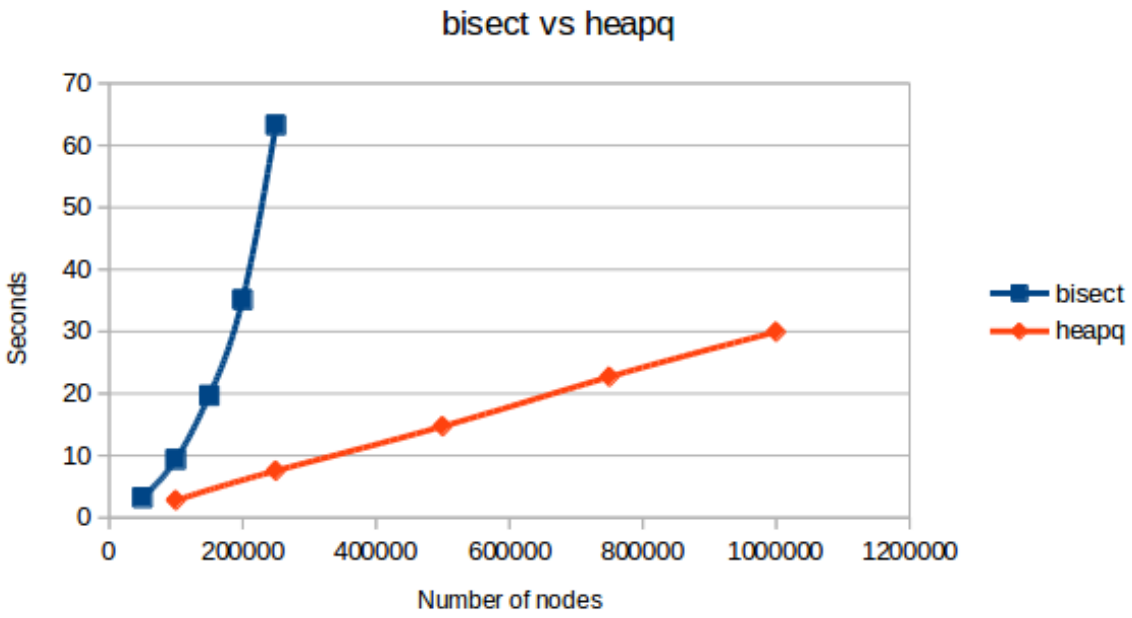
Several times it was needed hard-reseting the PC in order to stop it.



As we can observe, the function follows a linear growth.

Conclusions

The comparative among the experimental result of both data structure is displayed bellow.



Bisect behaves considerably worse; doubling the execution time of **heapq**, it has hardly managed to insert not even a quarter of the number of nodes **heaqp** has been able to, and by the graph we can deduce that gap will heighten very quickly as time goes on.

In conclusion, it is clear that we will be implementing our fringe by means of the **heapq** library, as it is faster in terms of magnitude than the **bisect** library.

Interesting parts of code

Firstly, we have ensured that our code is equivalent to the pseudocode described in the requeriment but working in Python 2.7. We can check it in the file /src/Problem.py, which is holding the methods **search** and **bounded_search,** they are equivalent to '**Búsqueda**' and '**Búsqueda_Acotada**', respectively.

def search(self, strategy, max_depth=10000, incr_depth=1):
current_depth = incr_depth
solution = None
while not solution and current_depth <= max_depth:
solution = self.bounded_search(strategy, current_depth)
current_depth = current_depth + incr_depth
 	
return solution
	Busqueda(Prob,estrategia,Prof_Max,Inc_Prof):Solución o Nada.
	Prof_Actual<-Inc_Prof
	Solución<- Ninguna
	Mientras (not Solución) and (Prof_Act=< Prof_Max):
	Solución= Busqueda_Acotada(Prob,estrategia,Prof_Actual)
	Prof_Actual <- Prof_Actual + Inc_Prof
	Fin_Mientras
	devolver Solución

Here in the table we can check that those methods are equivalent. We are aiming to return a solution or nothing (None in Python) and we need to do a while checking if there is not any solution, and if so we need to check if we can go deeper. We will one stop the loop when a solution or the maximum deep are reached.

This method is in charge of keep inserting nodes in the frontier and retrieving the one with the minimum value until a goal state is achieved.

```
def bounded_search(self, strategy, max_depth=10000):


    self.frontier = []
    initial_node = Node_Tree(self.initial_state)

    heapq.heappush(self.frontier, initial_node)
    solution = False

    while not solution and len(self.frontier) <> 0:
        current_node = heapq.heappop(self.frontier)
        if self.isGoal(current_node.state):
            solution = True
        else:
            successors_list = self.state_space.getSuccessors
            (current_node.state,  self.hash_table)
            successor_nodes = self.create_nodes(successors_list,
                                current_node, max_depth, strategy)
            for node in successor_nodes:
                heapq.heappush(self.frontier, node)


    if solution:
        return self.create_solution(current_node)
    else:
        return None
```

```
Búsqueda_Acotada (Prob,estrategia,Prof_Max): Solución o
NO_Solución
  #Proceso de inicialización
  frontera <- crear_frontera_vacia()
  n_inicial<- crea_nodo(padre<-Nada,estado<-
  Prob.EstadoInicial(),prof<-0,costo<-0,valor<-0)
  frontera.insertar(n_inicial)
  solución<- Falso
  #Bucle de búsqueda
  Mientras No solución y No frontera.esVacia() hacer
     n_actual<-selecciona_nodo(frontera)
     si Prob.Estado_Meta(n_actual.Estado) entonces
        solución<-True
   si_no
        LS<-Prob.Sucesores(n_actual.Estado)

        LN<-CreaListaNodosArbol (LS,nodo_actual,
                                 Prof_Max,estrategia)
        frontera.insertaLista(LN)
   fin_si
  Fin_Mientras
  #Resultado
  si solución entonces
        devolver CreaSolución(n_actual)
  si_no
        devolver NO_Solución
  fin_si
Fin_Búsqueda
```

We have implemented a method called '**getSuccesors()**', that corresponds to '**Prob.Sucesores**' in the pseudocode, which provides all the successor of a given state, which are tuples *(action, state, cost)*.

Then we pass that list to the **create_nodes()**, which is the one really in charge of creating the nodes which we will be inserting on the frontier. In here, we create a node from the different variables of its parent and then grant it with a value according to the search algorithm strategy we are currently using as it is shown in the following pseudocode.

switch (strategy)
case BFS: value = depth (parent_node) + 1
case DFS: value = - (depth (parent_node) + 1)
case DLS: value = - (depth (parent_node) + 1)
case IDS: value = - (depth (parent_node) + 1)
case UC: value = cost(current_node)

As we cannot implement a *switch* in Python, we had to nest several *elif*.

The method '**createSolution**' keeps retrieving the parent of a node from the goad node to the initial node (parent=null) and storing them into a FIFO queue.

Using the program

Should you want to execute this program, a Makefile has been written for this purpose.

Being at the root directory of this project, execute $make:

- build: to download the .osm file of the map of Ciudad Real (~3MB) and apply the appropriate permissions to the executable
- test: for executing the program with a right node
- test_error: for executing the program with an invalid node
- clean: for removing the map you have previously downloaded

Should you wish to execute a custom search, you need to type on a console:

python src/main.py <InitialNode> <latMin> <lonMin> <latMax><lonMax> <objetiveNode1>… <objetiveNodeN>

Example of use:

python src/main.py 3753271186 -3.9719000 38.9650000 -3.8847000 39.0062000 3753271182 3753271186

Bibliography

https://docs.python.org/2/library/bisect.html

https://docs.python.org/2/library/heapq.html

https://docs.python.org/2/library/stdtypes.html

https://docs.python.org/2/library/operator.html

https://docs.python.org/2/tutorial/datastructures.html

http://stackoverflow.com/questions/19979518/what-is-pythons-heapq-module

http://stackoverflow.com/questions/4713088/how-to-use-git-bisect