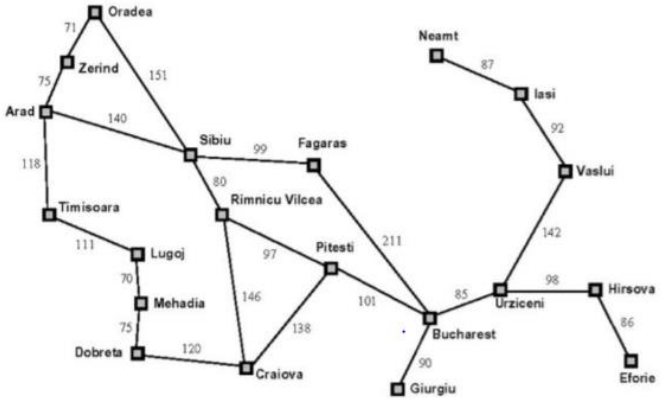
Graph Search Ramificación y Acotación vs Ramificación y Acotación con subestimación

A la hora de recorrer un grafo en concreto, tenemos distintas formas de recorrerlo, algunas mejores y otras peores. En este caso los 2 métodos por los que optamos a recorrer este grafo, serian Ramificación y acotación, y el método de Ramificación y Acotación con subestimación.

El grafo por recorrer es este:



Gracias a este programa, podremos saber que método de los 2 en este caso es mejor en cuanto a los saltos necesarios hasta llegar al resultado final.

Para implementar la búsqueda de Ramificación y Acotación, haremos uso la función graph_search, al cual le pasaremos una cola en concreto que cumpla las condiciones del método a estudiar.

Esta cola la peculiaridad que tiene es que una vez insertado los nodos en esta cola (lista abierta), se ordenan según su path_cost para que el de la posición inicial, sea el de menor path_cost y lo puedas extraer con el pop.

```
def bab(problem):
    return graph_search(problem, bab_queue())
```

```
class bab_queue(Queue):

    def __init__(self):
        self.A = []
        self.start = 0

    def append(self, item):
        self.A.append(item)

    def __len__(self):
        return len(self.A) - self.start

    def extend(self, items):
        self.A.extend(items)
        self.A.sort(key=lambda x: x.path_cost)

    def pop(self):
        e = self.A[self.start]
        self.start += 1
        if self.start > 5 and self.start > len(self.A) / 2:
            self.A = self.A[self.start:]
            self.start = 0
        return e
```

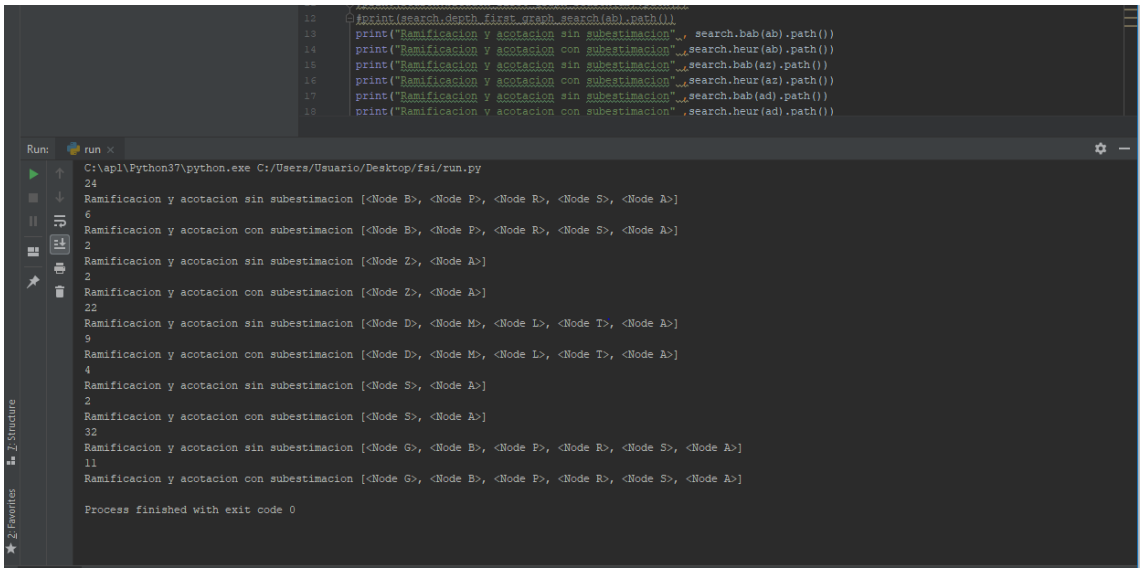
```
def graph_search(problem, fringe):
    closed = {}
    fringe.append(Node(problem.initial))
    count=0
    while fringe:
        count+=1
        node = fringe.pop()
        if problem.goal_test(node.state):
            print(count)
            return node
        if node.state not in closed:
            closed[node.state] = True
            fringe.extend(node.expand(problem))
    return None
```

En cuanto al método de búsqueda de Ramificación y Acotación con subestimación, también hacemos uso de la función graph_search, pero en este caso la lista la ordenamos por el path_cost de cada nodo, que es el path_cost acumulado del camino escogido hasta llegar a ese nodo y la suma de éste, con la heurística de este problema que es la distancia desde el nodo actual hasta el nodo destino.

```
def heur(problem):
    return graph_search(problem, heurQueue(problem))
```

```python
class heurQueue(Queue):

    def __init__(self, GPSProblem):
        self.A = []
        self.start = 0
        self.GPSProblem = GPSProblem

    def append(self, item):
        self.A.append(item)

    def __len__(self):
        return len(self.A) - self.start

    def extend(self, items):
        self.A.extend(items)
        self.A.sort(key=lambda x: (x.path_cost + self.GPSProblem.h(x)))

    def pop(self):
        e = self.A[self.start]
        self.start += 1
        if self.start > 5 and self.start > len(self.A) / 2:
            self.A = self.A[self.start:]
            self.start = 0
        return e
```

Para probar la diferencia entre estos 2 métodos, sacamos por pantalla los resultados de 5 pruebas, en las que cada en cada prueba se ejecutan los 2 métodos de búsqueda para así poder ver finalmente, como los 2 eligen el mismo camino, pero que donde se ve la diferencia, es en el numero de saltos que ha necesitado hacer cada una de las búsquedas, el cual representamos encima de cada resultado de camino.



Podemos observar que el método de Ramificación y Acotación con subestimación es notablemente mejor en cuanto a los saltos requeridos para obtener la solución.