Mauricio Soroco

60785250

Mathematics 441 – Portfolio 5

October 2022

Branch and Bound Explained and Applied to the Travelling Salesman Problem

Branch and Bound:

Branch and bound is a search algorithm that is optimal (meaning will return the optimal solution if it finds a solution) but not optimally efficient (doesn't search the least possible number of nodes). Branch and bound is used particularly for graphs that are deep and have a high branching factor since this strategy has a smaller space complexity compared to some other optimally efficient algorithms.

Algorithm:

```
Branch_and_bound(G, start, goal):
    Initialize stack S
    UB = ∞
    solution = NULL
    Add ⟨start⟩ to S
    While S is not empty:
        Select and remove path ⟨n_0,⋯,n_k⟩ from S
        If f(⟨n_0,⋯,n_k⟩) < UB:
            If goal(n_k):
                UB = path_cost(⟨n_0,⋯,n_k⟩)
                solution = ⟨n_0,⋯,n_k⟩
            For every neighbour n of n_k:
                If n ∉ ⟨n_0,⋯,n_k⟩
                    Add ⟨n_0,⋯,n_k,n⟩ to S
    Return solution
```

The neighbour relationship defines the graph `G`.

`Start` is the node with which we start the search.

The `goal` function defines what constitutes a solution.

The `f` function returns the "predicted" cost of the path (see below for more details).

A stack is a "last in, first out" data structure. The ones added more recently on the stack will be selected and removed first.

Heuristics:

A search heuristic $h(n)$ is an estimate of the cost of the lowest-cost path from node n to a goal node Thus the heuristic of a path $h(\langle n_0, \cdots, n_k \rangle) = h(n_k)$. The heuristic at a goal node is always 0.

We say that a heuristic $h(n)$ is admissible if it never is an overestimate of the minimum actual cost from n to a goal node. Hence an admissible $h(n)$ is a **lower bound** on the cost of getting from n to the nearest goal. This property is very important for the algorithm to work well.

Now the function `f()` is defined:

$$\mathrm{f}(\langle n_0, \cdots, n_k \rangle) = cost(\langle n_0, \cdots, n_{k-1} \rangle) + h(n_k) \tag{0}$$

And so, $\mathtt{f}(\langle n_0, \cdots, n_k \rangle)$ will always be an underestimate (due to admissibility of $h(n)$) for the minimum cost of the path $\langle n_0, \cdots, n_k \rangle$:

$$\mathrm{f}(\langle n_0, \cdots, n_k \rangle) \leq \mathrm{Path\_cost}(\langle n_0, \cdots, n_k \rangle) \tag{1}$$

Using this cost function `f`, we can be sure that a path such that $\mathtt{UB} \geq \mathrm{f}(\langle n_0, \cdots, n_k \rangle)$ will thus also satisfy:

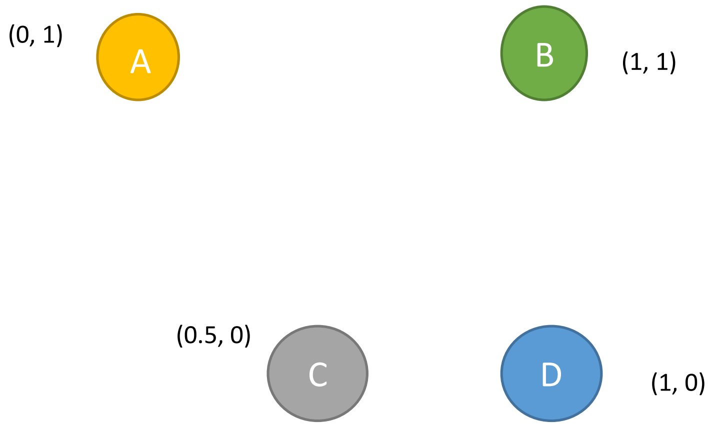
$$\mathrm{UB} \geq \mathrm{f}(\langle n_0, \cdots, n_k \rangle) \geq \mathrm{Path\_cost}(\langle n_0, \cdots, n_k \rangle) \tag{2}$$

And so there will be no point searching further along this path since it can only do as well as the current solution cost of `UB`.

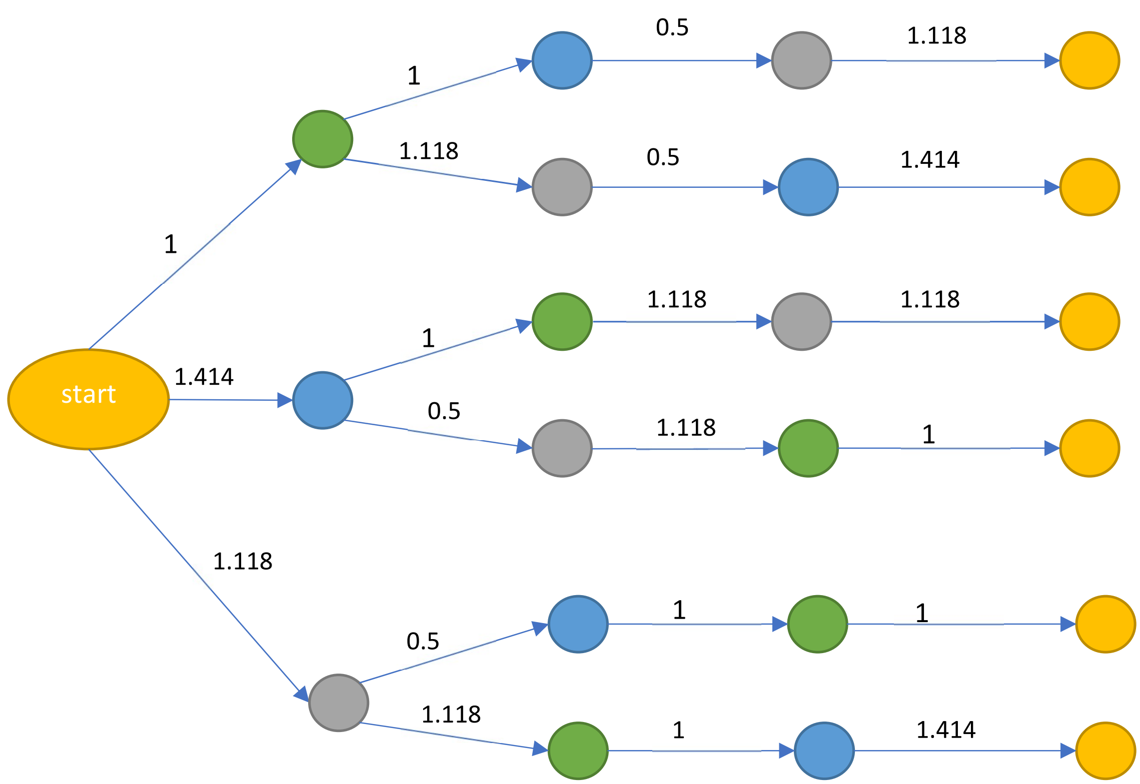
In short, branch and bound will only traverse paths that are lower in cost than the current path (see line 7 in code). As soon as a candidate path reaches or surpasses the cost of the current path there is no point continuing to search further along it.

The Travelling Salesman Problem:

Consider the Travelling Salesman Problem. Given a set of nodes, we are trying to find a cycle through all the nodes that has the minimum distance. Consider the following toy example:



In the TSP, we are in search of a cycle that passes through all the nodes, so it does not matter which node we start at. Let A be defined as the start node. We can convert the visualize the set of solutions to the TSP as the following graph:



Where the edge costs are proportional to the distance between the nodes.

NOTE: we duplicated the start node with leaf nodes so that valid solutions return to it as the last node. Thus, for the TSP every goal node is a copy of the start node.

What is left is to define an admissible heuristic for this problem.

Let the heuristic be:

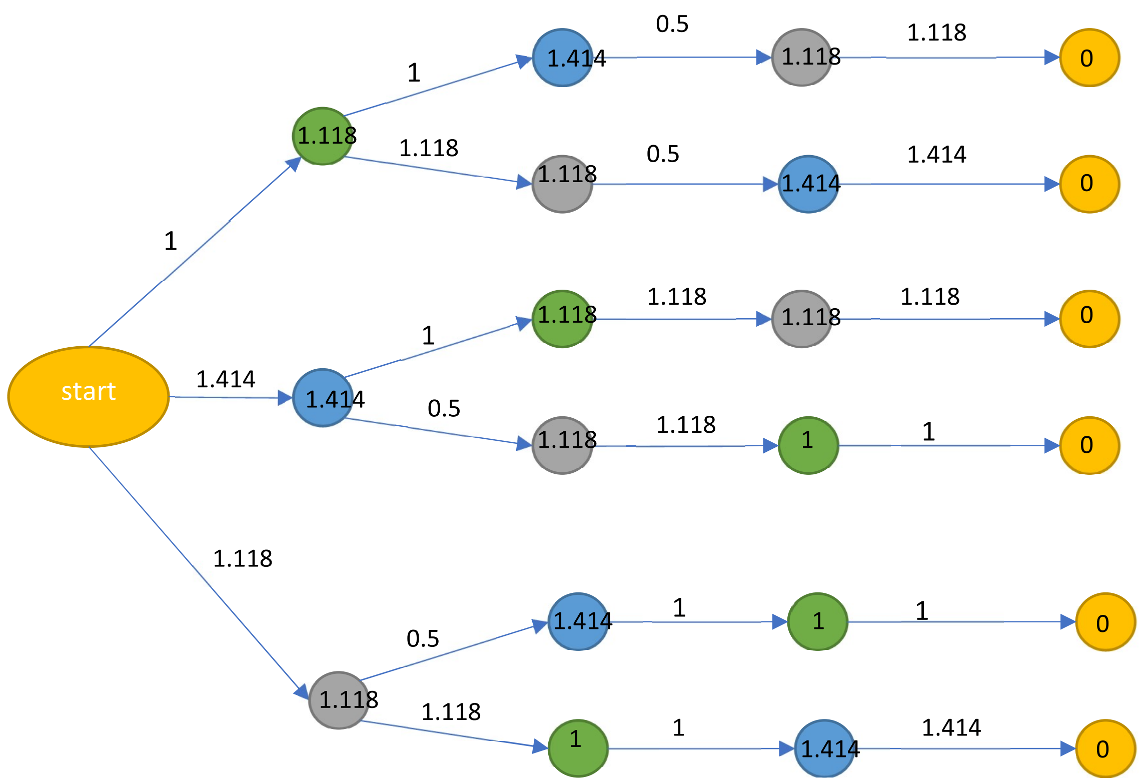
$$h(n) = \max\big(cost(n,\ \ n_i)\big) \qquad \forall \text{ remaining neighbours } \ n_i \text{ of } n \tag{3}$$

Where $cost(n, n_i)$ is proportional to the distance from n to n_i.

We claim that this is admissible for the following reasons. For any node that is remaining, we need to incur a cost to visit that node eventually (the start node included). If the start node is the only remaining node, then the heuristic at node n is exactly the maximum cost from n to a goal node (since it's the only cost from n). If there are multiple nodes that remain to be visited, then the distance to visit just the furthest one, $h(n)$, is less than the distance of any path that travels through all the remaining nodes due to the triangle inequality. Thus, the heuristic is admissible since:

$$h(n) \le \max(cost\big(\langle n, n_{i_1}, \cdots, n_{i_k}\rangle\big) \quad \text{where } \ k \text{ nodes remain} \tag{4}$$

Depicting the heuristics in our problem, we have:



Now we are ready to run B&B:

First, we add the start node to the stack (lines 1-4):

UB =	∞; solution = NULL
Stack:	
Path p	f(p) = cost(p)+h(p)
A	

We remove "start" from the stack and add its neighbours:

UB =	∞
Stack:	
Path	f(path)
AB	2.118
AD	2.828
AC	2.236

AC was added to the stack last. Select and remove AC and add its neighbours

UB =	∞
Stack:	
Path	f(path)
AB	2.118
AD	2.828
ACD	3.032
ACB	3.236

ACB was added most recently to the stack. We remove it from the stack and add its neighbours:

UB =	∞
Stack:	
Path	f(path)
AB	2.118
AD	2.828
ACD	3.032
ACBD	4.65

Remove ACBD and add its neighbours:

UB =	∞
Stack:	
Path	f(path)
AB	2.118
AD	2.828
ACD	3.032
ACBDA	4.65

This time when we remove a path from the stack (ACBDA), we find that it's a goal node. Thus, we update UB:

UB =	4.65 (ACBDA)
Stack:	
Path	f(path)
AB	2.118
AD	2.828
ACD	3.032

ACD is the last path on the stack. Remove it, and since its f-score is less than the UB add its neighbours.

UB =	4.65 (ACBDA)
Stack:	
Path	f(path)
AB	2.118
AD	2.828
ACDB	3.618

ACDB is the last path on the stack. Remove it, and since its f-score is less than the UB add its neighbours

UB =	4.65 (ACBDA)
Stack:	
Path	f(path)
AB	2.118
AD	2.828
ACDBA	3.618

ACDBA is the last path on the stack. Remove it. We evaluate it since its f-score is less than the UB and this time find that it is a goal node. Update UB (line 8-10 in code):

UB =	3.618 (ACDBA)
Stack:	
Path	f(path)
AB	2.118
AD	2.828

Now we continue as above:

UB =	3.618 (ACDBA)
Stack:	
Path	f(path)
AB	2.118
ADB	3.532
ADC	3.032

UB =	3.618 (ACDBA)
Stack:	
Path	f(path)
AB	2.118
ADB	3.532
ADCB	4.032

This time we remove ADCB from the stack. We note that it's f-score is already higher than UB. Since the f-score is an underestimate for the actual minimum path cost we can be sure that following this path will not lead to a better solution. Thus, we remove ADCB without adding its neighbours nor checking it's a goal node (line 7 in code).

The next item to remove is ADB:

UB =	3.618 (ACDBA)
Stack:	
Path	f(path)
AB	2.118
ADBC	4.65

This time we remove ADBC from the stack. We note that it's f-score is already higher than UB. Since the f-score is an underestimate for the actual minimum path cost we can be sure that following this path will not lead to a better solution. Thus, we remove ADBC without adding its neighbours nor checking it's a goal node (line 7 in code).

The next item to remove is AB:

UB =	3.618 (ACDBA)
Stack:	
Path	f(path)
ABD	3.414
ABC	3.236

UB =	3.618 (ACDBA)
Stack:	
Path	f(path)
ABD	3.414
ABCD	4.032

ABCD does not meet the UB condition for expanding this node. Continue:

UB =	3.618 (ACDBA)
Stack:	
Path	f(path)
ABDC	3.618

ABDC does not meet the UB condition for expanding this node. Remove it and continue:

UB =	3.618 (ACDBA)
Stack:	
Path	f(path)

Since there are no more elements on the stack, we are done (line 5 in code). We return the solution we found, namely ACDBA. (Note: as we saw in class, this solution path stays to the outside of all the nodes).

Since we are doing depth first search (with an upper-bounds check) this lets us find a solution as quickly as possible (since we know that all the goal nodes are at the deepest level of the graph). Depth first search will explore deeper nodes in a graph before it explores nodes in the same level. This lets us set the UB early on letting us prune later paths explored.

We demonstrated the key function of branch and bound which is to prune unpromising paths via the use of an admissible heuristic. In general, the better the heuristic approximates the actual path (while still be admissible) the better the performance of the algorithm.