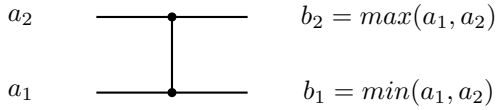
Algorithmics (ALG) Assignment

2-6 October 2023

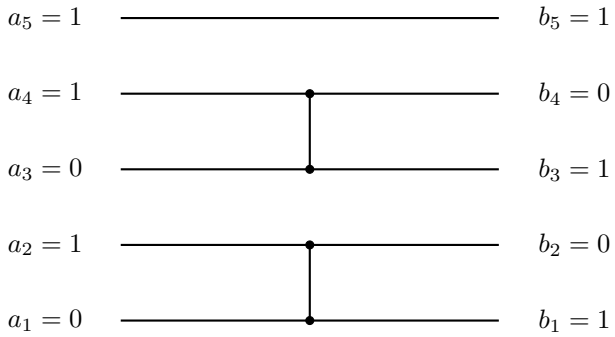
Total Number of Pages: 24

1.1 a) Can you construct a comparison network (not necessarily a sorting network) that maps the input $a_1 = 0$, $a_2 = 1$, $a_3 = 0$, $a_4 = 1$, $a_5 = 1$ to the output $b_1 = 1$, $b_2 = 0$, $b_3 = 1$, $b_4 = 0$, $b_5 = 1$?

In the lecture slides (slide 248 [1]) we define the comparator such that the max goes to the top wire and the min goes to the bottom wire.



Given such a comparator we have no way to achieve the desired output because we can only bubble the 1s up and not down making it impossible to get a 1 in the b_1 location. However, if we define the comparator the other way around i.e. $b_2 = min(a_1, a_2)$ and $b_1 = max(a_1, a_2)$ then we can achieve the desired result

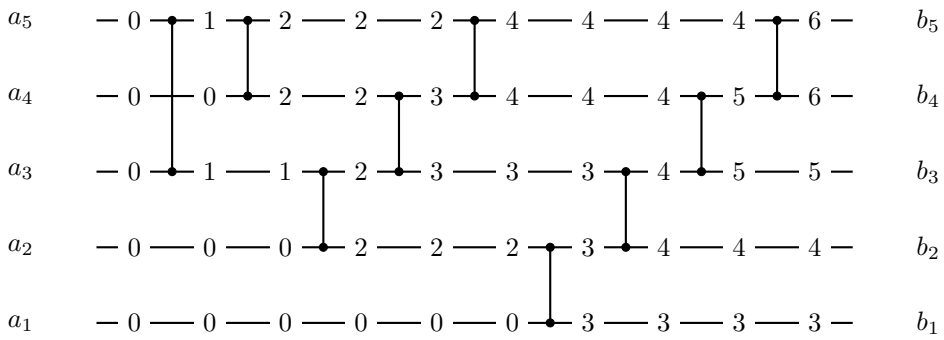


Where the comparator between a_3 and a_4 sinks the larger value 1 to b_3 and similarly the comparator between a_1 and a_2 sinks the 1 to b_1. We leave the a_5 untouched.

1.1 b) Determine the size and depth of each network

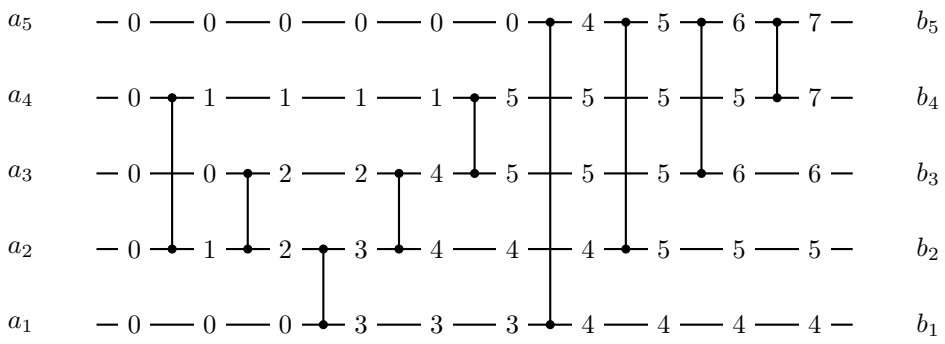
The size of a network is the number of comparators. The depth is the maximum number of comparators on any path through the network.

(i)



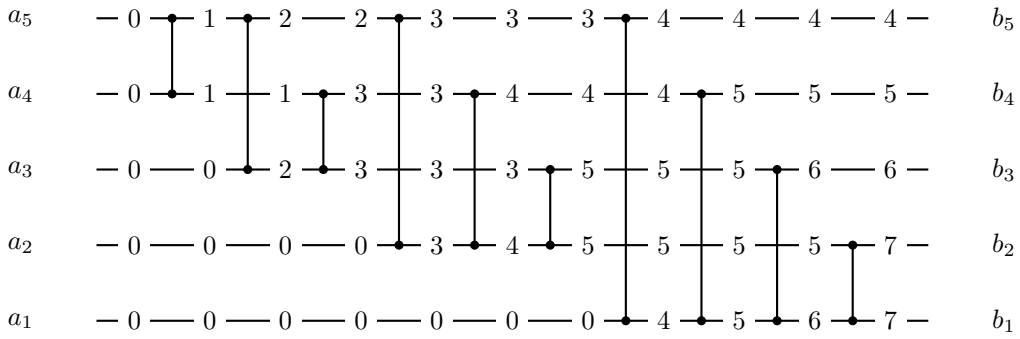
The size is 9 and the depth is 6.

(ii)



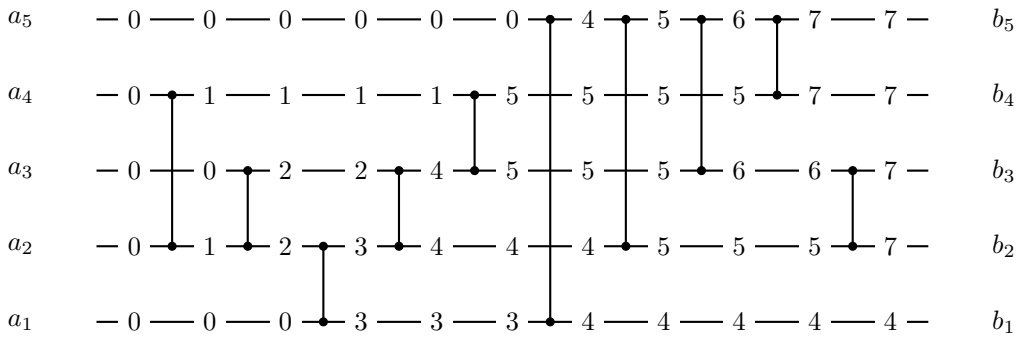
The size is 9 and the depth is 7

(iii)



The size is 10 and the depth is 7

(iv)



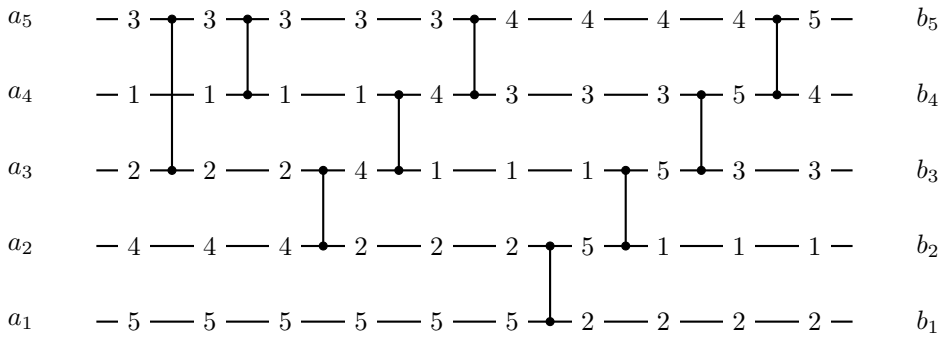
The size is 10 and the depth is 7

1.1 c) Which networks are sorting networks?

For the following networks we assume that the comparator is defined such that the maximum goes to the top wire and that the minimum goes to the bottom wire. Note that in the following diagrams the values do not represent depths but the ordering of the elements after each comparator.

(i)

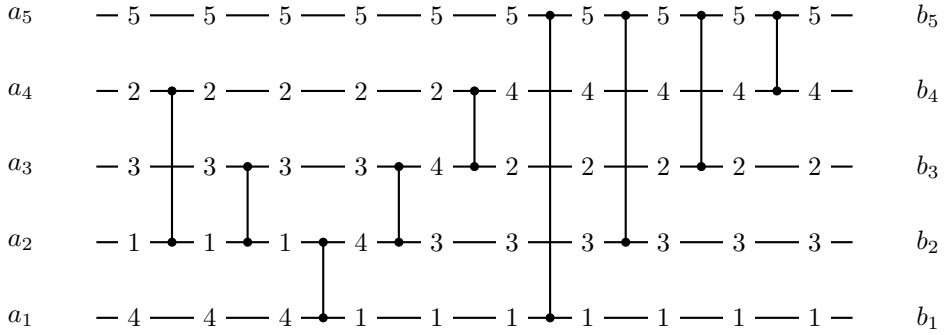
We can prove that this is not a sorting network by providing a counter-example. Given the input values 5, 4, 2, 1, 3, we would expect the output values to be 1, 2, 3, 4, 5



However, as we can see the first two comparators fail to sort 2, 1, 3 correctly from which the subsequent comparators can never recover even though we correctly bubble the 4 and 5 to the top. Thus the result is 2, 1, 3, 4, 5 has the 1 and 2 are incorrectly sorted.

(ii)

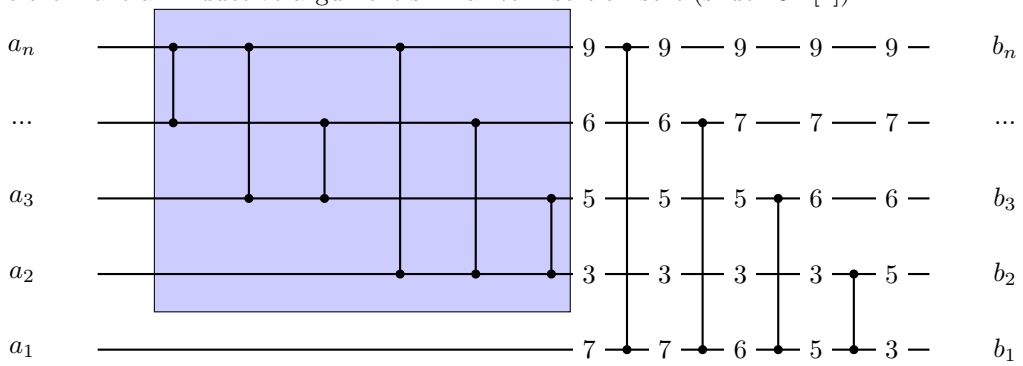
We can prove that this is not a sorting network by providing a counter-example. Given the input values 4, 1, 3, 2, 5 we would expect the output values to be 1, 2, 3, 4, 5.



However, as we can see we can again choose a 3 number combination, this time 1, 3, 2 which is not sorted after the first few comparators. And then because the 4 needs to be bubbled up and because the 5 is larger than any other number there are no comparators to sort the 1, 3, 2 correctly and we end up with 2 and 3 incorrectly sorted.

(iii)

This network has a recursive structure of comparing the new element to every line above it. We can therefore make an inductive argument similar to insertion sort (slide 252 [1])



For this network we can use an inductive approach to prove that it can sort.

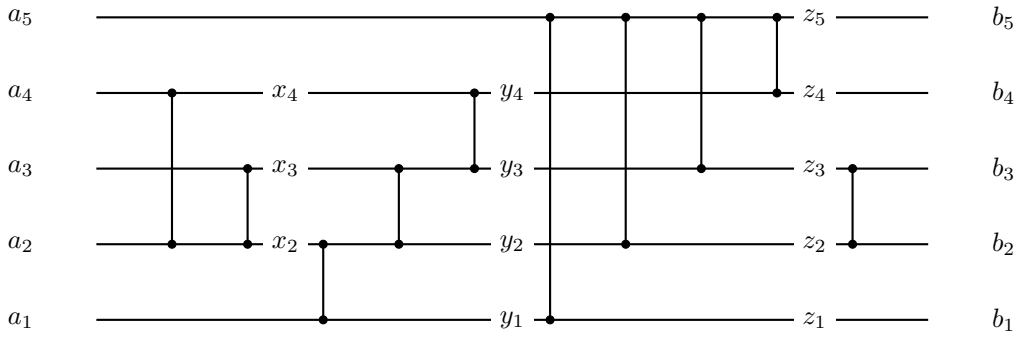
The base case with $n = 1$ is a straight line with a single element that by definition is already sorted.

In the case that $n > 1$ we make the inductive assumption that the previous $n - 1$ elements are sorted where in the diagram these are the $a_2 - a_n$ elements. In the diagram we have the sorted elements 9, 6, 5, 3 and the new element to be sorted 7 as an illustrative example to help explain the proof. We then have a set of $n - 1$ comparators between the new element a_1 and all previous elements. This is done in order where we compare with the largest first from a_n to a_2. If the new element a_1 is the minimum

then it will not move. If a_1 is greater than a_i where $2 \leq i \leq n$ then a_1 will swap there putting a_i on the bottom row, but we already know that all the previous elements were sorted so we will have a cascading effect where all subsequent elements will be shifted down one thus sorting the elements thus proving our inductive step and completing our proof by induction. This can be seen in the diagram where after the first comparison the 7 does not move as it is smaller than 9.

(iv)

We use proof by exhaustion, considering all possible orderings to show that this is a sorting network.



After the first two comparators we can guarantee that whatever is in x_2 is the minimum of a_2, a_3, a_4 because we effectively take $x_2 = \min(\min(a_2, a_4), a_3)$. However, we have no guarantees of the relative ordering of x_3 and x_4 as $x_4 = \max(a_2, a_4)$ which does not take account of a_3 necessarily.

The next 3 comparators cover the inputs $a_1 - a_4$. Given there are 3 numbers to compare to there are four possibilities (two are considered in the final bullet-point) for the relative size of a_1

- $a_1 < x_2$: Because a_1 is the new minimum it stays where it is. We then compare x_2 and x_3 again but these are left unchanged. However, we then compare x_3 and x_4 which may have been out of order but now are guaranteed to be in order. Thus the entire sequence y_1, y_2, y_3, y_4 is in order.

- $a_1 > x_2$ and $a_1 < x_3$: In this case the first comparator swaps the a_1 and x_2 where x_2 is the new overall minimum. However, as $a_1 < x_3$ we keep them in-place. This means there is no value left to bubble up and we can sort x_3 and x_4 in the final comparator of this section. Thus we know that the ordering y_1 as the minimum and y_4 as the maximum but we are not sure on the relative ordering of y_3 and y_4. It is for example possible with the input 3, 1, 4, 2 the the ordering after 5 comparators is 1, 3, 2, 4

- $a_1 > x_2$ and $a_1 > x_3$: As in the previous case x_2 becomes the new minimum. However, now we assume that $a_1 > x_3$ so we end up with the order from smallest to largest of x_2, x_3, a_1 where we know the relative ordering of x_2 and x_3 from the previous step. However, we did not know whether $x_3 > x_4$ or $x_4 > x_3$ so we need to consider both cases. If $x_3 > x_4$ then $a_1 > x_3$ must also be $a_1 > x_4$ therefore we have the ordering x_2, x_3, x_4, a_1 where x_3 and x_4 just like in the previous section are in the wrong order. If $x_4 > x_3$ i.e. they are in the correct order then a_1 may be greater or smaller than x_4 but either way we have a comparator to order them so the entire sequence y_1, y_2, y_3, y_4 is in order.

Thus we have established, by exhaustion, that y_1 is the minimum, y_4 is the maximum but it is possible y_2 and y_3 are out of order.

Next we need to consider the next four comparators which each compare a_5 with the other y_i values. As with the inductive argument in part iii there are five scenarios:

- $a_5 < y_1$: Here a_5 is the new minimum and y_1 is shifted to the top row. However, we know y_1 is the minimum of the y_is so we then end up with the ordering after the second comparator a_5, y_1, y_3, y_4, y_2. We then compare y_2 and y_3 (which we were not sure about the ordering of) and then compare the larger of the two with y_4 which we know is the biggest and therefore z_1 to z_5 are ordered correctly.

- $a_5 > y_1$ and $a_5 < y_2$: In this case, the first comparator does nothing, but the second comparator results in the ordering y_1, a_5, y_3, y_4, y_2 where y_2 has moved to the top row because it is larger. Then by the same argument we again compare y_2 and y_3 to order them correctly before comparing with y_4 so again we can conclude z_1 to z_5 are ordered correctly.

- $a_5 > y_1$, $a_5 > y_2$ and $a_5 < y_3$. In this case, the first and second comparators do nothing so we have the ordering y_1, y_2, y_3, y_4, a_5. We are not sure about the relative ordering of y_2 and y_3 so it is possible that $a_5 < y_3$, if this is the case they swap with the third comparator and the larger is then swapped with y_4 so again all z_1 to z_5 are ordered correctly

- $a_5 > y_1$, $a_5 > y_2$, $a_5 > y_3$ and $a_5 < y_4$. In this case, the first three comparators do nothing and the final one ensures that $z_5 > z_4$. Because we do not change the relative ordering of y_2 and y_3 it is possible that z_2 and z_3 are out of order as well.

- $a_5 > y_i, \forall i \in \{1-4\}$: In this case a_5 stays where it should and we can conclude that the z_is are all in order except z_2 and z_3 which might be out of order because y_2 and y_3 might have been.

Thus there are only two possible scenarios for the z_is. Either they are correctly in order or z_2 and z_3 are out of order. But we have one final comparator to fix this. Thus we have shown by exhaustion that this network is a sorting network.

1.1 d) Given a comparison network with n inputs how could you approach the general problem of deciding whether the network is a sorting network? How complex is this decision problem? Think about possible certificates. If helpful, consider the input elements are 0 or 1 - does this simplify the problem? Does it restrict the generality of your findings?

In the lecture slides[1] and the previous questions we use a number of approaches to prove that a comparison network is a sorting network including proof by induction for insertion and selection sort and even a single input for transposition networks. However, these approaches only work for specific networks and do not work in general, for example you cannot use proof by induction if the network does not have an inductive structure.

The question of whether a comparison network is a sorting network is a decision problem because the answer is either a yes or no. Proving no is relatively easy: you just need to provide a counter-example input (of course, coming up with the counter-example might not be!). This is known as having a short-certificate, which itself can be checked in polynomial time (technically $\Theta(m)$ in the number of comparators m although one could theoretically construct a network of arbitrary size for any input n). Proving yes is not easy however, one approach is to simply try every possible input where there are $n!$ permutations in an n-wire network however this factorial running time $\Theta(n!)$ (slide 485 [1]) is not polynomial but super-polynomial or unreasonable.

We can reduce the number of test cases to 2^n by only considering 0 and 1 input elements as suggested in the question. This 0-1 principle says that 'if an oblivious compare-exchange algorithm correctly sorts all input sequences consisting of only 0s and 1s, then it correctly sorts all inputs containing arbitrary values'[2]. This simplifies the problem without reducing the generality of the findings. We can prove this by equivalently proving the contrapositive by the fact that $P \to Q \Leftrightarrow \neg Q \to \neg P$. Suppose that a comparison network fails to sort some arbitrary input then we will show it will also fail to sort some $0-1$ input. Let the incorrectly sorted arbitrary input be $\{A_i \mid i = 1, \ldots, n\}$ where A_w is the smallest value in the incorrect place and A_l is the location where A_w should have gone. Then we can define a set of zeros and ones where $\{Z_i \mid i = 1, \ldots, n\}$ where $Z_i = 0$ if $A_i \leq A_w$ and $Z_i = 1$ if $A_i > A_w$. For example if the arbitrary array is $\{4, 1, 3, 2\}$ and it is incorrectly sorted to $\{1, 2, 4, 3\}$ then $A_w = 3$ is the smallest value in the incorrect place and $A_l = 4$. We can then define a set of 0-1s where $\{1, 0, 0, 0\}$ where because $A_l > A_w$ in our example $4 > 3$ we therefore have $Z_l > Z_w$ which are 1 and 0 respectively.

We then make an inductive argument (adapted in entirety from [3]) to prove that an because our function for defining Z_is is monotonically increasing we maintain the incorrect ordering from our arbitrary input that is incorrectly sorted, mathematically that after j comparisons it is possible to maintain the invariant that $A_i > A_w \leftrightarrow Z_i > Z_w = 0$. This is clearly true for the base case of 0 comparisons because this is exactly how we defined Z_i in terms of A_i. To prove the inductive step we assume the inductive assumption for j operations and consider the $j+1$ operation case. Suppose we have a comparison between i_1 and i_2 where $i_1 < i_2$ then there are four possible scenarios:

- $A_{i_1} \leq A_{i_2}$ and $Z_{i_1} > Z_{i_2}$: This is not possible because if $Z_{i_1} > Z_{i_2} \Rightarrow Z_{i_1} = 1$ and $Z_{i_2} = 0$ but by our inductive assumption that means that $Z_{i_1} > Z_w = Z_{i_2} \Rightarrow A_{i_1} > A_w \geq A_{i_2}$ but this contradicts $A_{i_1} \leq A_{i_2}$

- $A_{i_1} \leq A_{i_2}$ and $Z_{i_1} \leq Z_{i_2}$: In both cases, no swap occurs but both the smaller numbers are in the lower index position and the inductive step holds.
- $A_{i_1} > A_{i_2}$ and $Z_{i_1} \leq Z_{i_2}$: In this scenario a swap will occur in A but not in Z. However, we know that we cannot have $A_{i_1} > A_w \geq A_{i_2}$ i.e. one larger and one smaller than A_w because that would lead to a contradiction in the Z_is by the inductive assumption. Therefore, either both A_is are larger or both smaller than A_w implying that both Z_is are 1 or 0 which means the inductive step holds for $j+1$
- $A_{i_1} > A_{i_2}$ and $Z_{i_1} > Z_{i_2}$: The final case to consider is when both i_1 are larger and so we get swaps in both. For there to be a swap in the Z_is there must be one that is 1 and the other 0 and by the inductive assumption therefore one of the A_is is greater than A_w and the other less than, and after the swap the inductive step holds.

Therefore, we have shown that the inductive step always holds. This mean if our comparison network has j operations then we have $l < w$ because we have chosen A_w to be the smallest number out of place, i.e. it should by definition have been in the lower index l position. But we have just proved that after j operations on Z input the inductive hypothesis holds and therefore if $A_l > A_w \leftrightarrow Z_l > Z_w = 0$ but therefore Z_l is 1 and Z_w is 0 with $l < w$ means that Z is not sorted.

However, even with this reduction determining if a comparison networking is a sorting network is still a co-NP complete problem[4]. An NP problem is one in which the decision problem has a short certificate for the yes, but the no is intractable. In our case, we have the opposite where the yes is intractable - we have to test all 2^n cases, but the no has a short-certificate hence why it is considered co-NP. The problem is co-NP complete because it is possible to reduce the problem to another co-NP problem which can be done with a special type of non-deterministic reduction called a $\leq$tsn-reduction[4]

1.2 Given a circular race-track with n pits, with $f(i)$ litres available at pit i, and to race from pit i to clockwise neighbour need $g(i)$ litres of petrol. Determine a pit from which it is possible to race a complete lap where a starting pit is guaranteed by $\sum_{i=1}^{n} f(i) = \sum_{i=1}^{n} g(i)$

1.2 a) Describe an algorithm that solves the problem

We begin by defining the input data where at each pit i and an f and a g value. In Golang, my chosen language to implement the algorithm, we can write this as a struct.

```
type pit struct {
    i int
    f int
    g int
}
```

We have two variables, the first is the pitStart index which we initialize to 1 - note that in Golang slices are indexed from 0 but we are told in the question that the pits are from 1 to n, and a second fuelInTank which represents the amount of fuel in the tank after each pit-stop.

The algorithm itself involves looping through the list of pits, in Golang this is a slice. At each pit stop we calculate the marginal amount of fuel that is added to the tank i.e. pit.f − pit.g, note that this can be negative. If the rolling sum of fuel in the tank becomes negative then we reset pitStart to the next index and set the fuelInTank to zero. Otherwise we add the netFuelI.

```
func searchStartPit(pits []pit) int {
    pitStart, fuelInTank := 1, 0
    for _, pit := range pits {
        netFuelI := pit.f - pit.g
        if fuelInTank+netFuelI < 0 {
            pitStart = pit.i + 1
            fuelInTank = 0
        } else {
            fuelInTank += netFuelI
```

```
        }
    }
    return pitStart
}
```

Using the given example data we can see that the algorithm starts at Pit 1 with 0 fuel and never goes negative (see appendix). Rotating the data we find the algorithm can still find the optimal pit (again see appendix) - in this case pit 8 consistent with the provided example data.

1.2 b) Show that your algorithm is correct by exhibiting a suitable invariant

The invariant is that if starting from pit x you can only get to pit z and not further because you have run out of fuel then starting from any intermediate pit y where $x \leq y < z$ you will also run out of fuel by pit z

To prove this we start with our assumption of running out of fuel at pit z which means that, by definition, $\sum_{i=x}^{z} f(i) < \sum_{i=x}^{z} g(i)$ i.e. net fuel between x and z, $n_x^z = \sum_{i=x}^{z} f(i) - \sum_{i=x}^{z} g(i) < 0$

Then before we consider starting at pit y we calculate the amount of net fuel in the tank between pit x and pit $y-1$ which is $n_x^{y-1} = \sum_{i=x}^{y-1} f(i) - \sum_{i=x}^{y-1} g(i) \geq 0$ because if it was negative we would have run out of fuel before z not at z.

Now if we consider the net amount of fuel in the tank between y and z and because $n_x^z = n_x^{y-1} + n_y^z$ we can re-arrange to get $n_y^z = n_x^z - n_x^{y-1}$. Plugging in we get

$$\sum_{i=y}^{z} f(i) - \sum_{i=y}^{z} g(i) = (\sum_{i=x}^{z} f(i) - \sum_{i=x}^{z} g(i)) - (\sum_{i=x}^{y-1} f(i) - \sum_{i=x}^{y-1} g(i))$$

However we have already said the first term in the brackets is negative $\sum_{i=x}^{z} f(i) - \sum_{i=x}^{z} g(i) < 0$ and the second term in brackets is positive $\sum_{i=x}^{y-1} f(i) - \sum_{i=x}^{y-1} g(i) \geq 0$ and a negative number minus a positive number is still a negative number. Thus $\sum_{i=y}^{z} f(i) - \sum_{i=y}^{z} g(i)$ is a negative number i.e. $\sum_{i=y}^{z} f(i) < \sum_{i=y}^{z} g(i)$ i.e. starting from pit y we will also run out of fuel.

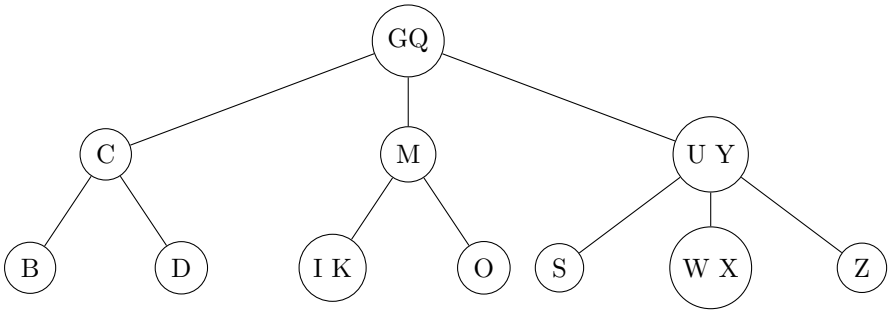
This invariant helps us write the algorithm because it means that we do not have to check every possible starting pit explicitly. Instead we can maintain a rolling sum and it goes negative we can immediately conclude that not only is pit x not a valid starting pit but so are all pits y up to pit z and therefore we update pitStart = pit.i + 1 (note this will never overflow because if the valid pitStart was **len**(pits) then this would be pit 0 which we have already tested).

1.2 c) Analyse the running time of your algorithm. Is your algorithm asymptotically optimal

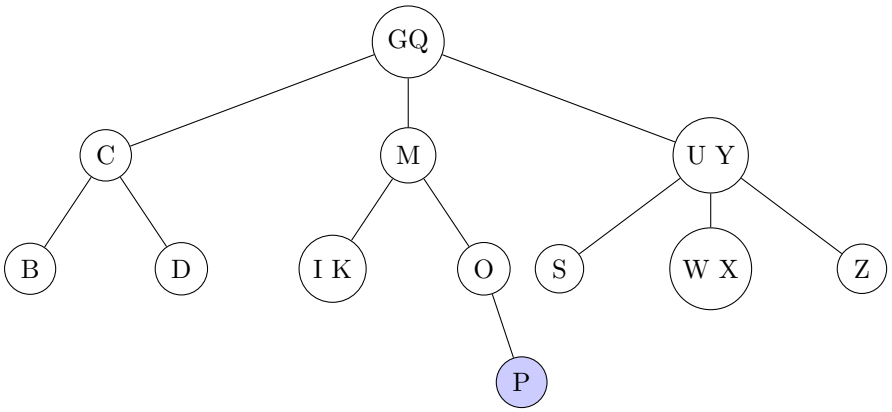
The upper bound for the running time for the algorithm in the worst-case is $O(1)$ for the initialization of pitStart and fuelInTank and then $O(n)$ for the for-loop over the list of pits of length n where all the operations like calculating netFuelI and updating fuelInTank are also $O(1)$. Therefore as $O(1) + O(n) = O(n)$ the algorithm has $O(n)$ i.e. linear complexity.

The lower bound in the worst case is still linear $\Omega(n)$ because we need to check all the pits. This can be proven by the fact that as the pits are arbitrarily ordered if our algorithm did not check every pit stop then we might miss the starting pit at that location making our algorithm invalid (technically we do not need to check that nth pit as if the first $n-1$ pits are not valid the nth must be because the $\sum_{i=1}^{n} f(i) = \sum_{i=1}^{n} g(i)$ however this does not change the complexity). Therefore as the lower and upper bounds are both linear we can say that the complexity is $\Theta(n)$

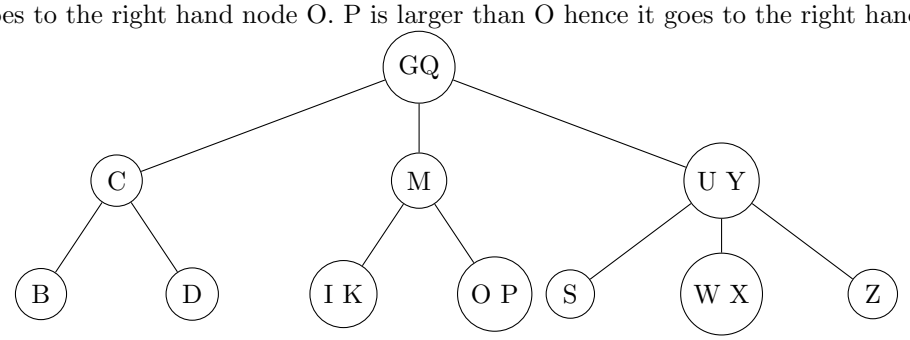
1.3 Consider the 2-3 search tree below:



1.3 a) Insert the element P into the tree

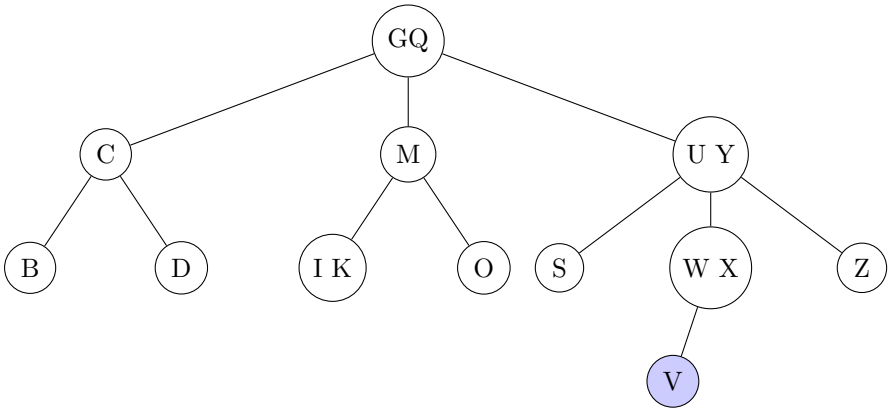


P is larger than G and less than Q hence it goes to the middle node M. P is larger than P hence it goes to the right hand node O. P is larger than O hence it goes to the right hand node again.

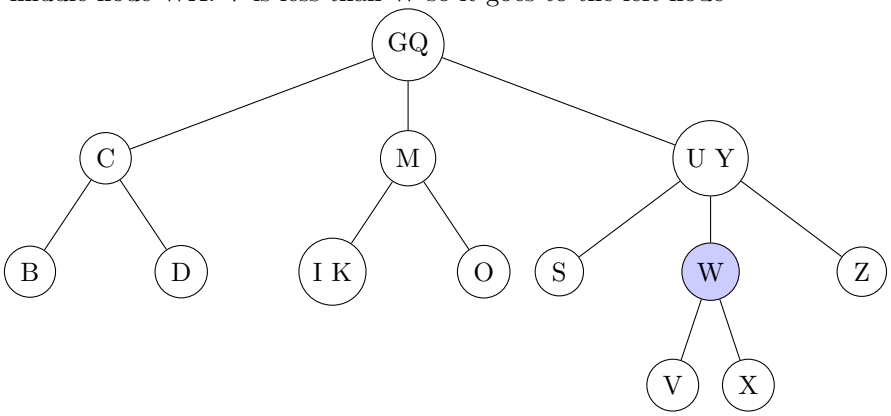


However, this violates the height condition so we repair the tree through the first local transformation (slide 367[1]) by turning the two-node O into a three-node with OP

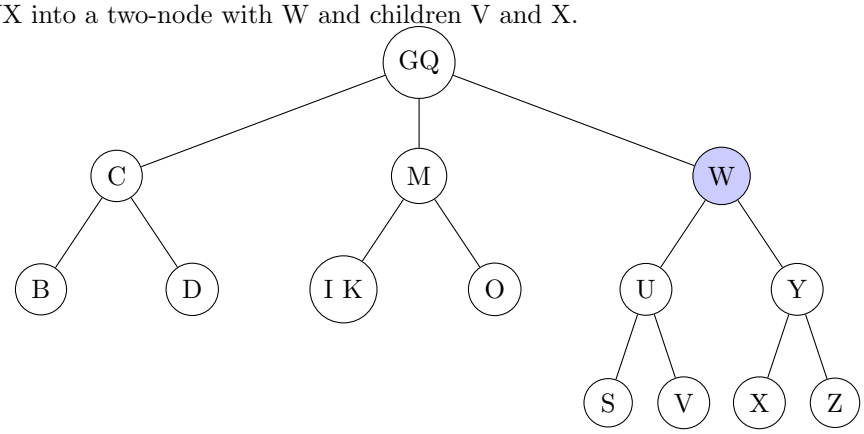
1.3 b) Insert the element V into the tree



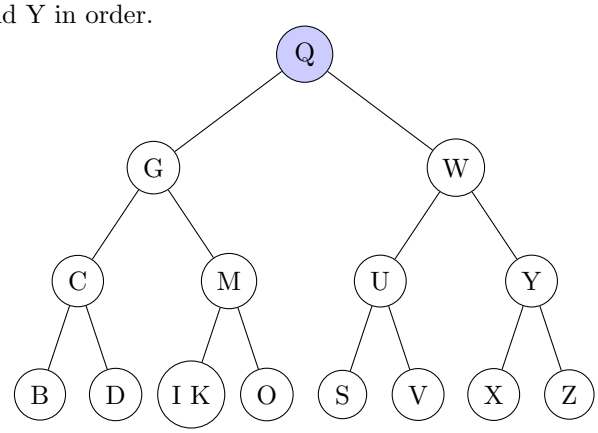
V is larger than Q so it goes to the right node UY. V is more than U but less than Y so it goes to the middle node WX. V is less than W so it goes to the left node



We repair through the second local transformation (slide 367[1]) by converting the parent three node WX into a two-node with W and children V and X.

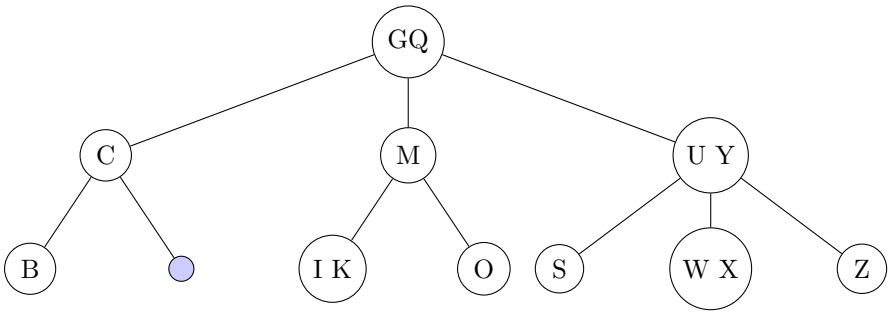


We again use the second local transformation (slide 367[1]) by converting the three node UY into a two node W where U and Y become the children and the children S, V, X, Z are distributed across U and Y in order.

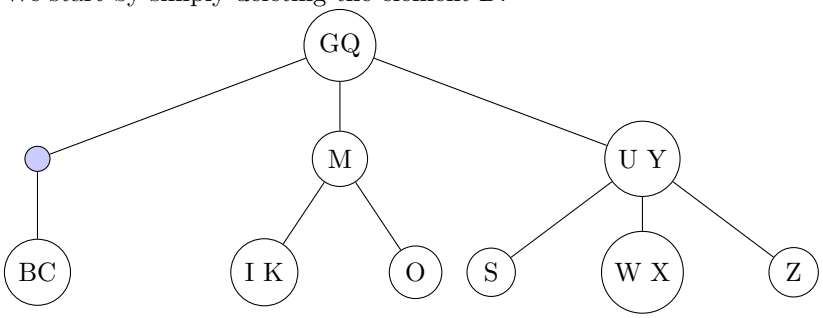


We again the second local transformation (slide 367[1]) except it is now reversed with the blue node on the right hand branch. We follow the same rule again where Q becomes the central node and G and W its children. At this point we have removed the height violation and we are done.

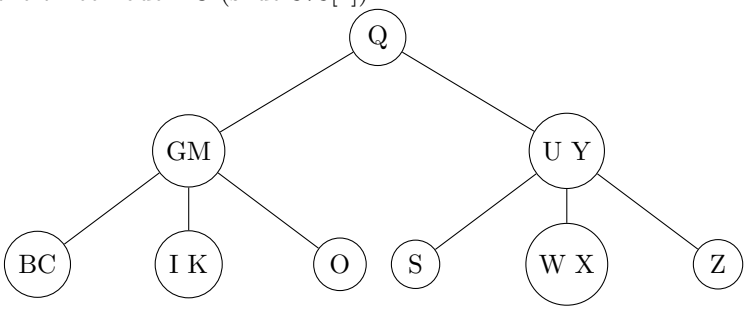
1.3 c) Delete the element D into the tree



We start by simply deleting the element D.

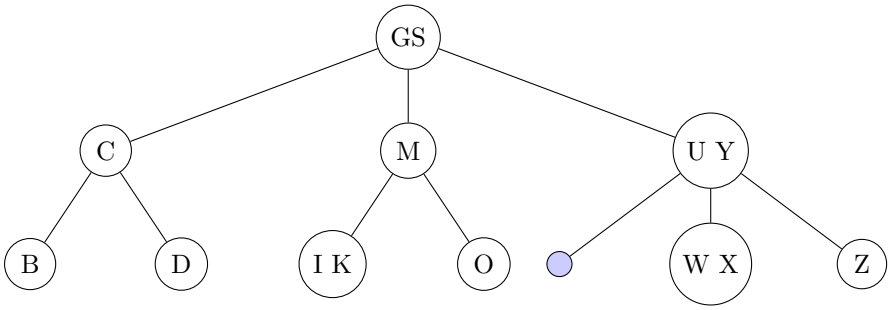


This creates the first local transformation with a two node parent C and a two node sibling B resulting in the three node BC (slide 373[1]).

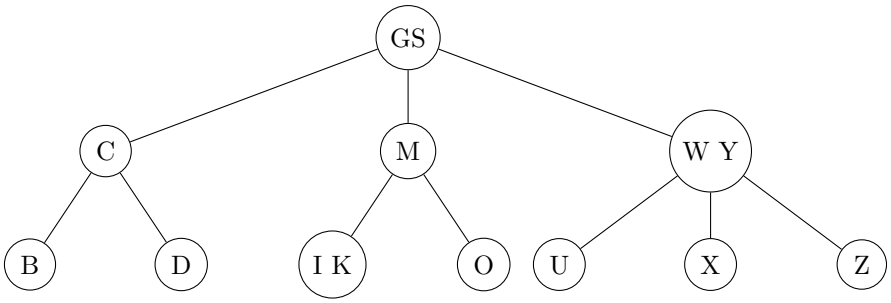


This now creates the scenario with a three node parent GQ and a two node sibling M which is the third local transformation (slide 374[1]) where GQ becomes Q and the blue node becomes GM taking on the two children of M: IK and O and we are done.

1.3 d) Delete the element Q into the tree



We delete Q and replace it with its inorder successor S (slide 372[1]) and in doing so create a blue node where S used to be.



When then have the fourth local transformation with a three node parent and three node sibling (slide 374[1]) where the UY node becomes WY creating U and X as children (along with Z).

1.4 You want to connect wind power plants to a north-south transmission line along a shortest route (either west or east). How should the location of the main transmission line so that the total length of the west-east lines is minimized? Write an essay of no more than 1000 words systematically applying different algorithmic methods. For each approach discuss the workings of the algorithm and estimate its running time. Does the problem become harder if the rigid geometric layout if relaxed?

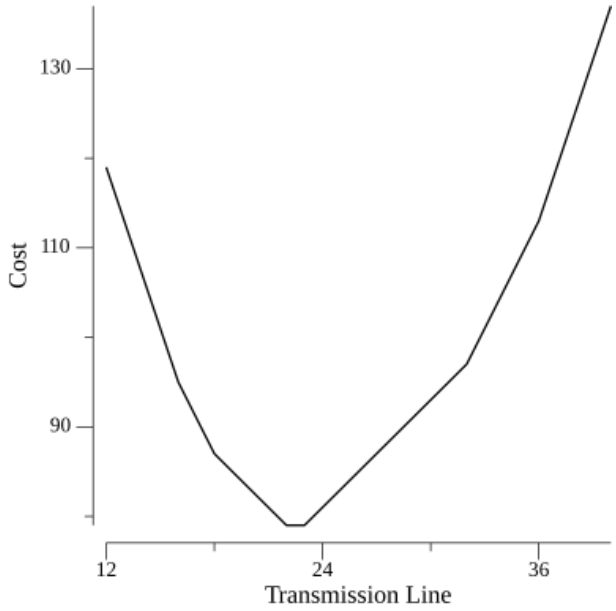
Word count for this section is 979.

Brute force

The brute-force approach is to try every possible transmission line location, calculate the total length of west-east lines and then to pick the lowest.

```
lowestCost, bestNorth, bestSouth := math.MaxInt, loc{}, loc{}
for x := mostWest; x <= mostEast; x++ {
    cost, north, south := calculateCost(powerPlants, x)
    if cost < lowestCost {
        lowestCost = cost
        bestNorth = north
        bestSouth = south
    }
}
```

This is illustrated in the code snippet above where we loop over all possible (in this case just integer) locations for the north-west transmission line x. The following graph illustrates the nature of the search where we find the optimal location of the tranmission line is from $\{x = 22, y = 38\}$ in the north to $\{x = 22, y = 22\}$ in the south for a total cost of 79 (including the north-south line which is fixed)



However, there are a few problems with this approach including only considering integer coordinates and the fact with n power plants and m the distance between the most west and east plants the complexity is $\Theta(mn)$ (technically $\Theta((m+1)n)$ if you include the $Theta(n)$ loop to look for the mostWest and mostEast plants.

Binary search

We can improve this by using binary search to find the local minimum. However, this is actually a global minimum because the cost function is unimodal which we can argue by 1. the absolute function which we use in our cost definition $C(x) = \sum_{i=1}^{n} |x - x_i|$ is convex by the triangle inequality 2. sum of convex functions is also convex 3. we can apply Jensen's inequality[5] to prove that the summation of absolutes is also convex.

Adapting binary search we calculate two midpoints at cost $\Theta(2n)$ and then choose to explore either mostWest-mid2 or mid1-mostEast thus for each round we reduce the search space by one third until the search space is less than our predefined epsilon eps which is logarithmic in $\Theta(\log m)$ (although not with base 2 as we are used to [1])

```
for mostEast - mostWest > eps {
    // search space: [mostWest - mid1 - mid2 - mostEast]
    mid1 := mostWest + (mostEast-mostWest)/3
    cost1, _, _ := calculateCost(powerPlants, mid1)
    mid2 := mostEast - (mostEast-mostWest)/3
    cost2, _, _ := calculateCost(powerPlants, mid2)

    // pick best search space
    if cost1 < cost2 {
        mostEast = mid2
    } else {
        mostWest = mid1
    }
}
```

Therefore the time complexity is $\Theta(n \log m)$. We find the same cost as the brute force approach even though we now consider float coordinates because there is a set of locations which give the same cost of 79. In this case we find the transmission line is from $\{x = 22.914, y = 38\}$ in the north to $\{x = 22.914, y = 22\}$ in the south.

Minimal spanning tree

If we relax the restriction of a grid-like pattern we can use the graph algorithm for a minimal spanning tree which finds the set of edges that both connects all power plants and has the minimal total length. There are two main approaches to this problem[6]:

- Kruskal's algorithm: Add the minimal edge that does not result in a cycle i.e. minimal before spanning as it is possible that the tree will not connect until the final edge.
- Prim's algorithm: Add the neighbouring vertex with the minimal edge i.e. spanning before minimal.

Both are greedy algorithms that exploit the Cut property[6] which says that for any cut C of the graph, if the weight of an edge E in the cut-set of C is strictly smaller than all other edges then it belongs to the minimal spanning tree of the graph.

In terms of implementation, Kruskal's requires we sort all the edges which costs $\Theta(E \log E)$ and then to use the union-find data structure[7] to track whether two subgraphs are connected and whether adding an edge will cause a cycle. The amortized time complexity of the find and union functions, when optimised with union by rank and path compression is $\Theta(O(1))$ on average[6].

For Prim's we need to use a heap in order to keep track of the minimal neighbour edge which in the case of a binary heap costs $\Theta(V + E)$ and $\Theta(\log V)$ time to extract the minimum element from the heap for an overall time complexity of $\Theta(E \log V)$. As both approaches use $\Theta(V)$ space we pick Prim's as the number of vertices is less than the number of edges in our case and therefore it is more optimal.

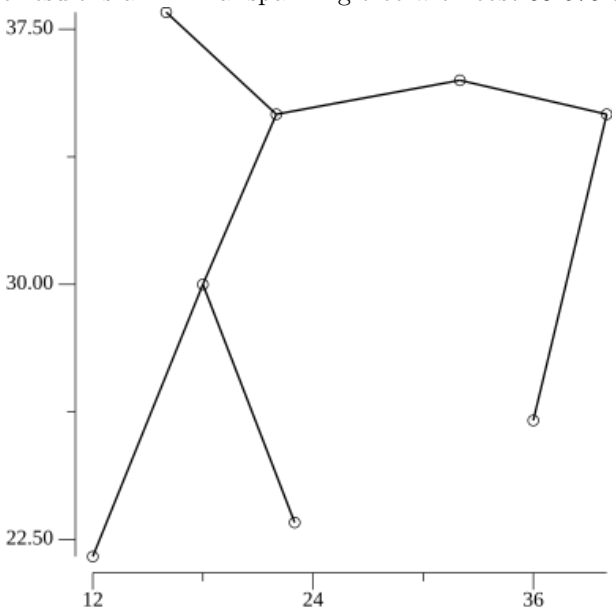
The key part of the algorithm is defining the min-heap which in Golang we can do by defining the key methods of the Heap interface which are Push, Pop and the the sort.Interface[8]

```
type Heap []edge

func (h *Heap) Push(x interface{}) { *h = append(*h, x.(edge)) }
func (h *Heap) Pop() interface{} {
        previous, n, popped := *h, h.Len(), edge{}
        *h, popped = previous[:n-1], previous[n-1]
        return popped
}
func (h Heap) Len() int           { return len(h) }
func (h Heap) Swap(i, j int)      { h[i], h[j] = h[j], h[i] }
func (h Heap) Less(i, j int) bool { return h[i].cost < h[j].cost }
```

And then we use the heap adding edges to vertices we haven't visited yet.

```
for h.Len() > 0 && visitedCount < len(powerPlants) {
    // get neighbour edge with lowest cost
    minEdge := heap.Pop(&h).(edge)
    // we are going from current tree to new vertex
    newVertex := minEdge.to
    if visited[minEdge.to] {
        continue
    }
    // process edge
    visited[newVertex] = true
    edges = append(edges, minEdge)
    visitedCount++
    rollingCost += minEdge.cost

    // add neighbours of new edge to heap
    for _, neighbourEdge := range adjList[newVertex] {
        if !visited[neighbourEdge.to] {
            heap.Push(&h, neighbourEdge)
        }
    }
}
```

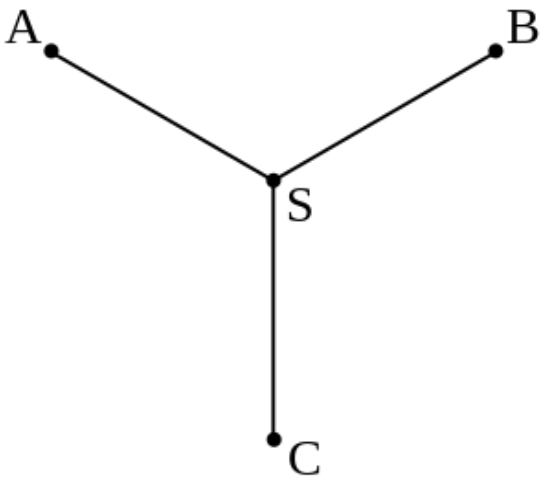
The result is a minimal spanning tree with cost 59.675 and the following shape.



Steiner tree problem

With the minimum spanning tree we assume that the only valid vertices are the locations of the current power plants and that edges can only exist between them. However, we can relax this condition by allowing additional non-terminal vertices to be added to the graph to create a Steiner tree, although this optimization is NP-hard[9].

If we consider the simpler geometric cases such as the solution for three points we can see that the steiner point S is located in the centre of the triangle (picture also from wikipedia[9]).



Inspired by this, we try to generate candidate non-terminal nodes by dividing the graph into Delaunay triangles. These triangles have the special property that they maximise the minimum of all the angles of the triangles[10], which is helpful for us because it makes their centroids of these triangles (their middle) potentially good non-terminal points[11]. These triangles are generated using a Golang package[12] and then for each triangle we manually calculate the centroid[13].

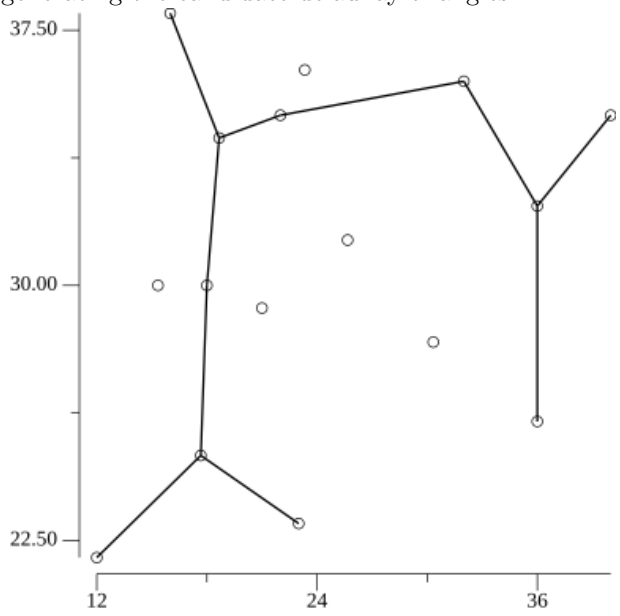
```
func nonTerminal(powerPlants []loc, w int, h int) []loc {
    // delauney triangulation
    powerPlantMap := make(map[loc]bool, len(powerPlants))
    var points normgeom.NormPointGroup
    for _, powerPlant := range powerPlants {
```

```
            points = append(points, normgeom.NormPoint{powerPlant.x, powerPlant.y})
            powerPlantMap[powerPlant] = true
        }

        // add non-terminal nodes
        nonTerminalNodes := make([]loc, 0)
        for _, triangle := range triangulation.Triangulate(points, w, h) {
            nonTerminal := centroid(triangle)
            if !powerPlantMap[nonTerminal] {
                nonTerminalNodes = append(nonTerminalNodes, nonTerminal)
            }
        }
        return nonTerminalNodes
}
```

And then for each possible non-terminal node we try adding it to the power plants nodes and see if it creates a smaller minimal spanning tree, keeping it if it does.

```
nTs := nonTerminal(powerPlants, 1, 1)
for _, nT := range nTs {
    cost, edges := prims(append(finalNodes, nT))
    if cost < minCost {
        fmt.Println(minCost)
        minCost = cost
        minEdges = edges
        finalNodes = append(finalNodes, nT)
    }
}
```

The result is the following Steiner tree where we have added three non-terminal nodes and reduced the cost of the network from 59.675 to 56.397. The complexity however has increased a lot because we loop over the non-terminal nodes and do Prim's algorithm on each for $\Theta(EV \log V)$, as well as the cost of generating the candidate delaunay triangles.



Appendix

1.2 a) Describe an algorithm that solves the problem

```
package main

import "fmt"

type pit struct {
    i int
    f int
    g int
}

func searchStartPit(pits []pit) int {
    pitStart, fuelInTank, totalNetFuel := 1, 0, 0
    for _, pit := range pits {
        netFuelI := pit.f - pit.g
        if fuelInTank+netFuelI < 0 {
            pitStart = pit.i + 1
            fuelInTank = 0
        } else {
            fuelInTank += netFuelI
        }
        totalNetFuel += netFuelI
    }
    if totalNetFuel < 0 {
        return -1
    }
    return pitStart
}

func printFuel(pits []pit, start int) {
    fuelInTank := 0
    for i := 0; i < len(pits); i++ {
        mI := (start + i - 1) % len(pits)
        pitI := mI + 1
        fmt.Printf("Pit %d: fuel in tank %d\n", pitI, fuelInTank)
        fuelInTank += pits[mI].f - pits[mI].g
    }
}

func main() {
    testPits := []pit{
        {i: 1, f: 9, g: 5},
        {i: 2, f: 1, g: 4},
        {i: 3, f: 9, g: 4},
        {i: 4, f: 5, g: 2},
        {i: 5, f: 7, g: 9},
        {i: 6, f: 3, g: 2},
        {i: 7, f: 2, g: 4},
        {i: 8, f: 6, g: 3},
        {i: 9, f: 1, g: 6},
        {i: 10, f: 2, g: 4},
        {i: 11, f: 7, g: 4},
        {i: 12, f: 0, g: 2},
        {i: 13, f: 1, g: 4},
```

```
    }
    start := searchStartPit(testPits) // pit 1
    printFuel(testPits, start)
}
```

Which prints out from pit 1 the following fuels in the tank

```
Pit 1: fuel in tank 0
Pit 2: fuel in tank 4
Pit 3: fuel in tank 1
Pit 4: fuel in tank 6
Pit 5: fuel in tank 9
Pit 6: fuel in tank 7
Pit 7: fuel in tank 8
Pit 8: fuel in tank 6
Pit 9: fuel in tank 9
Pit 10: fuel in tank 4
Pit 11: fuel in tank 2
Pit 12: fuel in tank 5
Pit 13: fuel in tank 3
```

Using the same data but shifting the location of the pits we find

```
    testPits2 := []pit{
        {i: 1, f: 2, g: 4},
        {i: 2, f: 6, g: 3},
        {i: 3, f: 1, g: 6},
        {i: 4, f: 2, g: 4},
        {i: 5, f: 7, g: 4},
        {i: 6, f: 0, g: 2},
        {i: 7, f: 1, g: 4},
        {i: 8, f: 9, g: 5},
        {i: 9, f: 1, g: 4},
        {i: 10, f: 9, g: 4},
        {i: 11, f: 5, g: 2},
        {i: 12, f: 7, g: 9},
        {i: 13, f: 3, g: 2},
    }
    start2 := searchStartPit(testPits2) // pit 8
    printFuel(testPits2, start2)
}
```

prints out a new starting pit 8 which is consistent with the provided data i.e. is successfully picking the pit for which the fuel in the tank never becomes negative

```
Pit 8: fuel in tank 0
Pit 9: fuel in tank 4
Pit 10: fuel in tank 1
Pit 11: fuel in tank 6
Pit 12: fuel in tank 9
Pit 13: fuel in tank 7
Pit 1: fuel in tank 8
Pit 2: fuel in tank 6
Pit 3: fuel in tank 9
Pit 4: fuel in tank 4
Pit 5: fuel in tank 2
Pit 6: fuel in tank 5
Pit 7: fuel in tank 3
```

1.4 Brute force

```
package main

import (
    "fmt"
    "math"

    "gonum.org/v1/plot"
    "gonum.org/v1/plot/plotter"
    "gonum.org/v1/plot/vg"
)

func bruteForce(powerPlants []loc) (int, loc, loc, []point) {
    // get most east and most west x values
    mostWest, mostEast := powerPlants[0].x, powerPlants[0].x
    for _, powerPlant := range powerPlants {
        if powerPlant.x < mostWest {
            mostWest = powerPlant.x
        }
        if powerPlant.x > mostEast {
            mostEast = powerPlant.x
        }
    }

    // try every possible transmission line and calculate the costs
    lowestCost, bestNorth, bestSouth := math.MaxInt, loc{}, loc{}
    data := make([]point, 0)
    for x := mostWest; x <= mostEast; x++ {
        cost, north, south := calculateCost(powerPlants, x)
        data = append(data, point{x, cost})
        if cost < lowestCost {
            lowestCost, bestNorth, bestSouth = cost, north, south
        }
    }
    return lowestCost, bestNorth, bestSouth, data
}

func calculateCost(powerPlants []loc, transmissionLine int) (int, loc, loc) {
    horizontalCost, verticalCost := 0, 0
    mostNorth, mostSouth := powerPlants[0].y, powerPlants[0].y
    for _, powerPlant := range powerPlants {
        // calculate north-south transmission line length
        if powerPlant.y > mostNorth {
            mostNorth = powerPlant.y
        }
        if powerPlant.y < mostSouth {
            mostSouth = powerPlant.y
        }
        // calculate east-west transmission line length
        horizontalCost += abs(powerPlant.x - transmissionLine)
    }
    verticalCost = mostNorth - mostSouth
    return horizontalCost + verticalCost, loc{transmissionLine, mostNorth},
        loc{transmissionLine, mostSouth}
}
```

```
func abs(x int) int {
    if x < 0 {
        return -x
    }
    return x
}

type loc struct {
    x int
    y int
}

type point struct {
    x    int
    cost int
}

func plotCosts(data []point) {
    p := plot.New()
    p.X.Label.Text = "Transmission Line"
    p.Y.Label.Text = "Cost"
    pts := make(plotter.XYs, len(data))
    for i, pt := range data {
        pts[i].X = float64(pt.x)
        pts[i].Y = float64(pt.cost)
    }
    line, err := plotter.NewLine(pts)
    if err != nil {
        fmt.Println(err)
    }
    p.Add(line)
    if err := p.Save(4*vg.Inch, 4*vg.Inch, "line_graph.png"); err != nil {
        fmt.Println(err)
    }
}

func main() {
    powerPlants := []loc{
        {12, 22},
        {16, 38},
        {18, 30},
        {23, 23},
        {22, 35},
        {36, 26},
        {32, 36},
        {40, 35},
    }
    cost, north, south, data := bruteForce(powerPlants)
    fmt.Println(data)
    fmt.Println(cost, north, south)
    plotCosts(data)
}
```

1.4 Binary search

We omit a lot of the tests and helper functions which we are largely the same as the brute force case.

```
func binarySearch(powerPlants []loc)
    (float64, loc, loc, []point, []point) {
    // get most east and most west x values
    mostWestInt, mostEastInt := powerPlants[0].x, powerPlants[0].x
    for _, powerPlant := range powerPlants {
        if powerPlant.x < mostWestInt {
            mostWestInt = powerPlant.x
        }
        if powerPlant.x > mostEastInt {
            mostEastInt = powerPlant.x
        }
    }

    // adapted binary search
    mostWest, mostEast := float64(mostWestInt), float64(mostEastInt)
    eps := 1e-1
    data1 := make([]point, 0)
    data2 := make([]point, 0)
    for mostEast-mostWest > eps {
        // search space: [mostWest - mid1 - mid2 - mostEast]
        mid1 := mostWest + (mostEast-mostWest)/3
        cost1, _, _ := calculateCost(powerPlants, mid1)
        mid2 := mostEast - (mostEast-mostWest)/3
        cost2, _, _ := calculateCost(powerPlants, mid2)
        data1 = append(data1, point{mid1, cost1})
        data2 = append(data2, point{mid2, cost2})
        // pick best search space
        if cost1 < cost2 {
            mostEast = mid2
        } else {
            mostWest = mid1
        }
    }
    lowestCost, bestNorth, bestSouth :=
        calculateCost(powerPlants, mostWest)
    return lowestCost, bestNorth, bestSouth, data1, data2
}
```

1.4 Minimal spanning tree

```
    package main

import (
        "container/heap"
        "fmt"
        "math"

        "gonum.org/v1/plot"
        "gonum.org/v1/plot/plotter"
        "gonum.org/v1/plot/vg"
)

func prims(powerPlants []loc) (float64, []edge) {
    // build visited map
    visited := make(map[loc]bool, len(powerPlants))
    // build adjacency list of edges
```

```
    adjList := make(map[loc][]edge, len(powerPlants))

    // initialize visited map and adjacency list
    for i, powerPlantI := range powerPlants {
        visited[powerPlantI] = false
        for j := i + 1; j < len(powerPlants); j++ {
            powerPlantJ := powerPlants[j]
            // from power plant I to power plant J
            fromIToJ := edge{powerPlantI, powerPlantJ,
                calculateEdgeCost(powerPlants, i, j)}
            adjList[powerPlantI] = append(adjList[powerPlantI], fromIToJ)
            // from power plant J to power plant I
            fromJToI := edge{powerPlantJ, powerPlantI,
                calculateEdgeCost(powerPlants, i, j)}
            adjList[powerPlantJ] = append(adjList[powerPlantJ], fromJToI)
        }
    }

    // initailize heap with point 0 edges
    pointZeroEdges := adjList[powerPlants[0]]
    h := make(Heap, len(pointZeroEdges))
    copy(h, pointZeroEdges)
    heap.Init(&h)
    visited[powerPlants[0]] = true
    visitedCount := 1
    rollingCost := 0.0

    // initialize data
    edges := make([]edge, 0)

    // loop until all power plants are visited
    for h.Len() > 0 && visitedCount < len(powerPlants) {
        // get neighbour edge with lowest cost
        minEdge := heap.Pop(&h).(edge)
        // we are going from current tree to new vertex
        newVertex := minEdge.to
        if visited[minEdge.to] {
            continue
        }
        // process edge
        visited[newVertex] = true
        edges = append(edges, minEdge)
        visitedCount++
        rollingCost += minEdge.cost

        // add neighbours of new edge to heap
        for _, neighbourEdge := range adjList[newVertex] {
            if !visited[neighbourEdge.to] {
                heap.Push(&h, neighbourEdge)
            }
        }
    }
    return rollingCost, edges
}

func calculateEdgeCost(powerPlants []loc, i, j int) float64 {
    sumSquares := math.Pow(powerPlants[i].x-powerPlants[j].x, 2) +
```

```
            math.Pow(powerPlants[i].y-powerPlants[j].y, 2)
    return math.Pow(sumSquares, 0.5)
}
```

1.4 Steiner tree problem

Most of the code is the same as the Prims case. The main additions being the function to generate the non-terminal points and the centroid calculation

```
func nonTerminal(powerPlants []loc, w int, h int) []loc {
    // delauney triangulation
    powerPlantMap := make(map[loc]bool, len(powerPlants))
    var points normgeom.NormPointGroup
    for _, powerPlant := range powerPlants {
        points = append(points,
            normgeom.NormPoint{powerPlant.x, powerPlant.y})
        powerPlantMap[powerPlant] = true
    }

    // add non-terminal nodes
    nonTerminalNodes := make([]loc, 0)
    for _, triangle := range triangulation.Triangulate(points, w, h) {
        nonTerminal := centroid(triangle)
        if !powerPlantMap[nonTerminal] {
            nonTerminalNodes = append(nonTerminalNodes, nonTerminal)
        }
    }
    return nonTerminalNodes
}

func centroid(triangle geom.Triangle) loc {
    centroid := loc{
        x: float64(triangle.Points[0].X+triangle.Points[1].X+
            triangle.Points[2].X) / 3.0,
        y: float64(triangle.Points[0].Y+triangle.Points[1].Y+
            triangle.Points[2].Y) / 3.0,
    }
    return centroid
}
```

References

[1] Ralf Hinze. Algorithmics, 2023. University of Oxford, MSc Software Engineering module slides.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2009.

[3] Andrew Lohr. Algorithmics. Rutgers University, Solutions to CLRS.

[4] Moon Jung Chung and B. Ravikumar. Strong nondeterministic turing reduction—a technique for proving intractability. *Journal of Computer and System Sciences*, 39(1):2–20, 1989.

[5] Wikipedia. Jensen's inequality.

[6] LeetCode. Graph explore card.

[7] Wikipedia. Union find.

[8] go.dev. Golang heap.

[9] Wikipedia. Steiner tree problem.

[10] Wikipedia. Delaunay triangles.

[11] Wikipedia. Euclidean minimum spanning tree.

[12] github.com/RH12503. Triangula.

[13] Wikipedia. Centroid.