CAB203 Project: Widget factory

James Wyatt – n8040478

1 Introduction

My company has recently built a factory to create final products (e.g., widgets) from a feed stock (e.g., PLP). The factory has several machines each of which has one process. This process takes an input and produces an output. The input of a machines process can be either the feed stock, which can be used to create a product (e.g., thingamabob) or can be a product itself (e.g., gadget). Whilst the output of a machines process can either be the final product (e.g., widget) or a product (e.g., whatchamacallit) which then can be used in a dissimilar process to create a different product (e.g., doodad). This newly produced product can then be used in a different machines process to create a final product (e.g., widget). Each machine has a capacity of how much output it can produce in kilograms per hour. Conveniently it requires 1KG of feed stock to create one product and thus one final product. Each machines process is not reversable, meaning there's a machine to create a widget from a gadget but not gadgets from widgets.

Table 1 – *(Source: CAB203 project file – Widgetsamplefile.csv)*

Machine	Process	Input	Output	Capacity
Telza-3000	Telza	gadget	widget	7
Gadgetron	Fredison	PLP	gadget	1
Thingamabob-X	Mosk	thingamabob	gadget	2
PtoT v2.0	Baber-Hosch	PLP	thingamabob	5
Qrussa i3 mk 7	FDM	PLP	whatchamacallit	4
Backyard composter	Anerobic Composting	whatchamacallit	doodad	3
Mach 17 CNC	Quotientive Manufacturing	doodad	widget	2
A big laser and some goo	SLA	thingamabob	whatchamacallit	2
Beetroot vapour extractor	Dye-casting	gadget	whatchamacallit	3

(The company has collected information about the machines, processes and capacities, as seen above.)

As each machine can be set to produce its output at any rate between 0 and its capacity. The goal is to determine the optimal setting for each machine to produce the maximum number of final products (e.g., widgets) per hour without any machine producing excess products (e.g., whatchamacallit). The factory will be able to manufacture other final products in the future, meaning the above table could be altered.

In this report we consider the arrangement above of machines and the process they have. The goal of the widget factory problem is, given a CSV file relating to these machines, deduce the feedstock and final product. Then find the maximum flow from the feed stock to the final product while considering the capacity and process of each machine. This creating a list of optimal settings (as seen below) for each machine to produce the maximum number of final products per hour without producing excess products. If it is found that there is no optimal setting, each machine should be set to 0.

{'Telza-3000': 4, 'Thingamabob-X': 2, 'Backyardcomposter': 3,…}

2 Problem

We begin by describing the problem rigorously. An arrangement of the input, output and capacity of all the machines given is called a configuration of the problem. We can first represent a configuration as a tuple of ordered pairs where *M (u, v)* is the input and output of each machines process. For example, the instance can be modelled by:

$$M = \{(\text{Input}, \text{Output})\}$$

$$M = \{(\text{gadget}, \text{widget}),\ (\text{PLP}, \text{gadget}),\ (\text{thingmabob}, \text{gadget}), \ldots(\text{gadget}, \text{whatchamacallit})\}$$

This showing a point-to-point connection between each input and output. From this we can show that the edge relationship between the unique input and output (vertices) as being the individual process and capacity a machine has. The notation of this traversal from an input to an output is modelled by *v1, e1, v2*, this is where *v1* originates from the input of a machine, *e1* represents the machine, process and capacity of that machine, and *v2* represents the correlating output of a machine, this can be modeled by:

$$(\text{gadget}, (\text{Telza-3000}, \text{Telza}, 7)\ , \text{widget})$$

We then need to identify which point-to-point to use in which order. To do this we must consider the feed stock as the source s vertex and the final product as the drain d vertex. To determine this, we must consider the symmetrical difference of each machines input as set A and output to set B, such that:

$$s = \{x : (x \in A) \oplus (x \in B), x \in A\}$$

$$d = \{x : (x \in A) \oplus (x \in B), x \in B\}$$

We can then consider the solved configuration as the maximum amount of traversals from s to d while considering the weight of each traversal between each of the vertices from s to d. As each edge has a certain capacity which can receive a certain amount of flow, this flow running through an edge must be less than or equal to the capacity. Such that, the flow is a map $f: E \to \mathbb{N}$ with a capacity constraint $f_{uv} \leq c_{uv}$ for all $(u, v) \in E$. This flow will then produce the optimal setting for each machine.

This giving our instance model, the problem can be present as a directed graph $G = (V, E)$. The set of vertices V is the set of unique elements of the individual input set A and the output set B. Such that:

$$V = A \cup B$$

Meanwhile, E will contain an edge for each pair in M.

$$E = M \cup \{(v, u) : (u, v) \in M\}$$

Furthermore, let w represent the capacity of each machine. As stated in the introduction the capacity is the maximum amount of output a machine can produce per hour. Note, the capacity of a machine should be over 0. This can be modeled by:

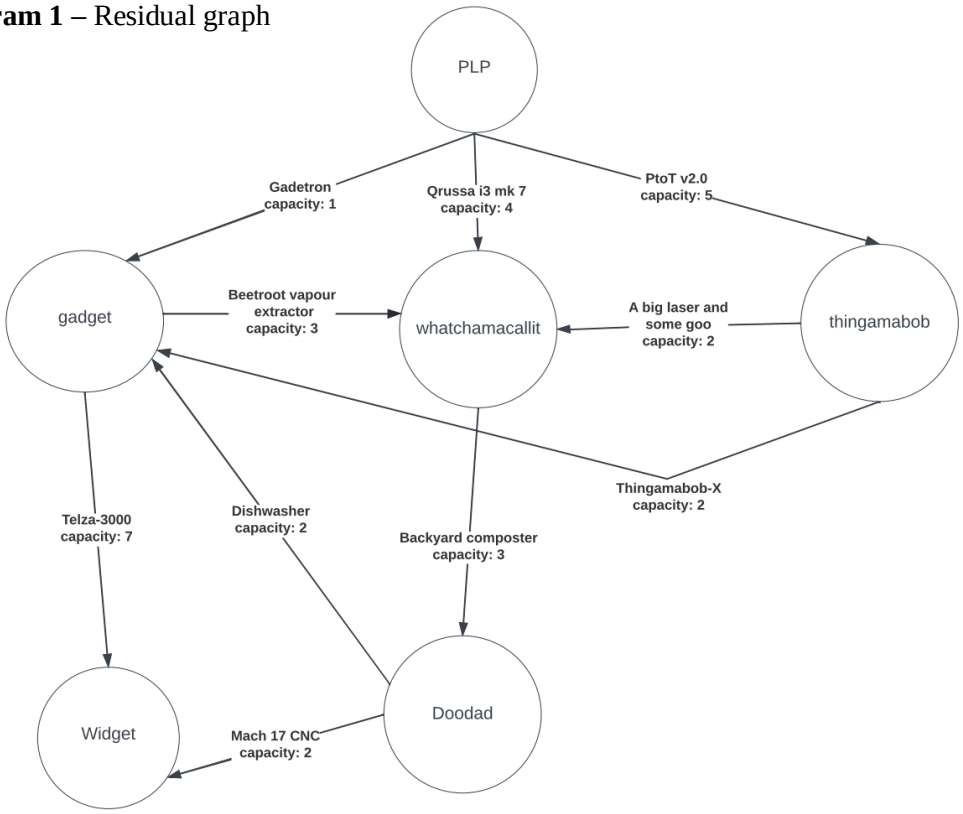
$$w: E \to \mathbb{N}$$

$$e \in E, \text{ the weight of } e \text{ is } w(e).$$

This giving each edge a weighting of the capacity for the given machine and a source and drain for our graph. From this we can determine that this is a weighted directed graph, with w, s and d. As stated in the introduction, a machines process cannot be reversable, from this an edge will

only point from vertices A to B and not B to A. This meaning the graph is anti-symmetrical. An example can be seen below:

Diagram 1 – Residual graph



From this we can summarize the widget factory problem as follows: find the flow f_{max} with maximum value from source s to drain d.

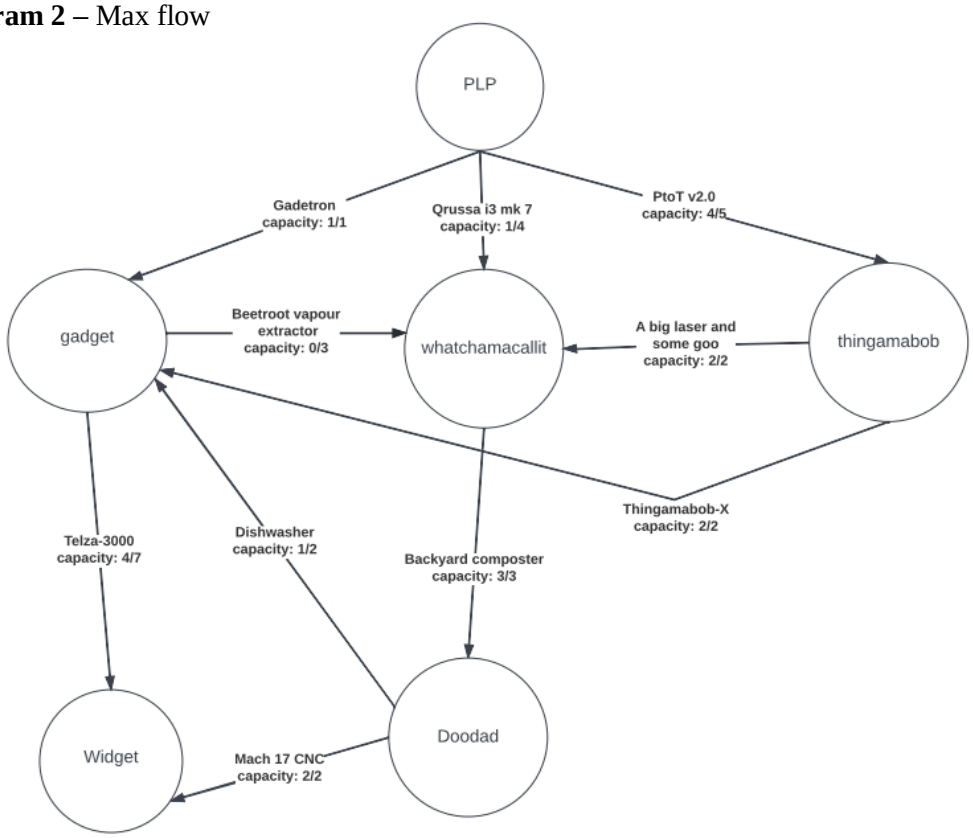
3 Solution

From this we can solve the maximum flow of the graph with *Ford-Fulkerson algorithm* and a *depth first search* using mostly similar concepts from CAB203. The standard method for finding the maximum flow, we first start at d and traverse the graph to s. As each edge has a certain capacity which can receive a certain amount of flow, we can consider each traversal from each vertex in the graph as a flow for that individual edge, which uses one unit of the capacity of that edge. Each time we traverse from the source to the drain we may not in-fact find the path which uses the max flow. Using the Ford-Fulkerson method we can create augmented paths, which is a sequence of vertices $s = V_1 V_2, \ldots, V_n = d$ where no vertex is repeated. Given a flow f and an augmented path with capacity w, a new flow t can be created with a higher value, such that:

$$t(u,v) := \begin{cases} f((u,v)) + w \text{ the path goes forwards along (u, v)} \\ f((u,v)) - w \text{ the path goes backwards along (u, v)} \\ f((u,v)) \text{ otherwise} \end{cases}$$

We can repeatedly find the augmented paths through the residual graph until no more augmented paths can be found, for any path from s to d using *depth first search*. Using this determines the maximum flow and can show which machine should have which setting to ensure that the no other machines will produce more than what is needed to create a final product. For any directed graph $G = (V, E)$ with source $s \in V$, drain $d \in V$, and weights w, the maximum value of flow on G is equal to the minimum value of a cut of G. This showing by taking the *min cut* on the graph between s and d, we should see $min\ cut = max\ flow$, thus ensuring a correct solution of f_{max}. Noted that there can be many variations of the path taken between the vertices to produce f_{max} , but a condition would be that what is coming from the source must be equal to what is going into the drain. If there is no such path, then the *depth first search* will indicate so. An example of this max flow can be seen below:

Diagram 2 – Max flow



4 Implementation

The implementation closely follows techniques used in CAB203, using the functions provided in *digraphs.py*. Although firstly we collect the data by reading the CSV file and converting the contents into a python dictionary. Four empty lists are then used to hold the machine, input, output and capacity in the given CSV file. The input and output lists are then converted into python sets; this is to ensure we only get the unique values in each of corresponding list. We can then find the symmetric difference between the input and output sets by using the inbuilt python function *symmetric_difference*. This producing the source vertex and the drain vertex, which can simply be deduced by the unique element in both input and output. This combined set is then checked against the original input and output lists to find which element is the correct source and drain. The inbuilt python function *union* is then used to find the union between the input set and

output set, this producing the vertices. *Zip_longest* is then used to combine the input and output lists into a 2-tuple and combined with the capacity list to create a new dictionary with the key and value: *((input, output) : capacity)*, this producing the edge between each vertex and the weight of each edge. This follows the conditions described in the Problem section. The vertices, edges with their weight, source and drain are then used with the *maxFlow* function from *digraphs.py*. The function *maxflow* is used to return the maximum flow for the given anti-symmetrical weighted directed graph. This is done by initially setting the flow of all the edges to 0. The *augmentingPath* function is then called repeatedly to find all the augmenting paths in the given graph. The *augmentingEdges* function is called within to return an edge set which represents the edge that can be in an augmenting path. The *findPath* function is then called to return the list of vertices forming a directed path from start to end, this function uses a simple *depth first search* algorithm. The base case is returned if $start = end$, then if a path is found it is then returned to *maxFlow*, if a path is not found the *findPath* returns none. The *augmentFlow* function is then called with the path that was returned as a parameter. This is used to return a valid flow along the augmenting path, this process continues until all valid augmenting paths have been explored. This follows the conditions described in the Solution section. The max flow is then returned, and a new dictionary is created to hold the relationship between the machine, input/output and the input/output and capacity.

The main function is *optimiseWidgets* which takes the starting configuration. It first builds the augmented paths using *maxFlow* as described above. The solved configuration f_{max} is then returned (otherwise None), a list comprehension is used to translate the sequence of the edges and their valid flow to a sequence of machines and their optimal setting.

5 Sources

Lecture 9, CAB203

project_report_sample.pdf, CAB203

digraphs.py, CAB203

Maximum flow problem - Wikipedia. (2022). Retrieved 16 May 2022, from https://en.wikipedia.org/wiki/Maximum_flow_problem

Ford–Fulkerson algorithm - Wikipedia. (2022). Retrieved 16 May 2022, from https://en.wikipedia.org/wiki/Ford%E2%80%93Fulkerson_algorithm