

LICEUL TEORETIC "AVRAM IANCU"

LUCRARE PENTRU OBTINEREA
COMPETENȚELOR PROFESIONALE ÎN
INFORMATICĂ

JobShop Scheduler

Profesor coordonator:
Prof Salantiu Crina

Realizator:
Croitoru Cristian

Cuprins

1	Introducere	4
2	Limbaje, Librării și Tehnologii folosite	4
2.1	Limbajul C++	4
2.1.1	Librării	4
2.2	Limbajul Python	4
2.2.1	Librării	4
2.3	Tehnologii folosite	5
2.3.1	Build Script	5
3	Structura Proiectului	6
3.1	Structura folderelor	6
3.2	Fișierele neclasificate	6
3.3	Folderul "src"	7
3.4	Folderul "include"	8
3.4.1	Subfolderul "gui"	8
3.4.2	Subfolderul "text2num"	8
3.5	Folderul "bin"	9
3.6	Folderul "scripts"	9
4	Parsare	9
4.1	Parsarea Fuzzy	9
4.1.1	Mașină de stare	10
4.1.2	Legendă mașină de stadiu	11
4.1.3	Descriere detaliată	12
4.1.4	Funcția text2num	17
4.2	Parsarea JSON	17
5	Modelarea Datelor	18
5.1	Clasa Part	20
5.2	Clasa Machine	21
5.3	Clasa Job	22
5.3.1	Structura de graf bipartit	22
5.3.2	Vectorii de parti și mașinării	23
5.4	Constructorii	23

6	Stocarea Datelor	24
7	Interfață	27
7.1	Funcționalitate	29
7.2	Poze	30
8	Descrierea Problemei	30
9	Îmbunătățiri pe viitor	31
9.1	Parsare	31
9.2	Programarea operațiunilor	31
9.3	Vizualizare	31
10	Generarea planului de producție	31
10.1	Construirea IDMap-ului	31
10.2	Graful de producție	32
10.2.1	Nodul Parte	32
10.2.2	Nodul Mașină	32
10.2.3	Procesul de legare	32
10.3	Construirea Dataframe-ului	32
10.4	Construirea Time Matrix-ului	32
10.5	Construirea cozilor și sortarea	33
11	Cerințe Software și Hardware	33

1 Introducere

Proiectul "Job-Shop Scheduler" constă într-o aplicație multi-platforma care le oferă companiilor și fabricilor o modalitate de a gestiona comenzile de produse, mașinăriile industriale deținute și de a genera automat planuri de producție eficiente folosind informația furnizată de ei.

2 Limbaje, Librării și Tehnologii folosite

Proiectul folosește două limbaje de programare, patru librării externe pentru cele două limbaje și diferite programe pentru compilarea și build-ul programului.

2.1 Limbajul C++

Limbajul principal al proiectului este C++, mai exact C++20. Pentru compilare s-a folosit compilatorul GNU GCC 12.2.1.

2.1.1 Librării

Am utilizat pentru salvarea fișierelor modificate de program librăria [JSON for Modern C++](#).

Interfață grafică este scris folosind librăria [wxWidgets](#), care permite crearea aplicatilor grafice multi-platforma (Windows, Mac, Linux).

2.2 Limbajul Python

Python este folosit pentru partea de vizualizare a datelor calculate de program pentru planul de producție. Motivul pentru care această operațiune nu este efectuată în C++ este deoarece Python are o multitudine de librării pentru vizualizarea datelor modelate prin structuri matematice cum ar fi vectori sau matrici.

2.2.1 Librării

Pentru vizualizare am folosit librăria [matplotlib](#), care se ocupă cu crearea graficelor și librăria [numpy](#) care permite modelarea datelor sub formă obiectelor matematice.

2.3 Tehnologii folosite

Fiind un proiect mare, cu mai multe fișiere sursă și cu librării externe, am avut nevoie de un Build System, adică un program pentru a gestiona fișierele sursă și pentru a crea un fișier "Makefile" care e ulterior folosit în compilarea paralelizată prin comandă "make" pe Linux și echivalentele ei din alte sisteme de operare.

Am decis să folosesc [Premake](#), un build system cu o interfață ușoară printr-un script LUA.

2.3.1 Build Script

```
dofile "use_wxwidgets.lua"

workspace "Atestat"
    configurations { "Debug", "Release" }

project "Atestat"
    kind "WindowedApp"
    language "C++"
    targetdir "bin/{cfg.buildcfg}"

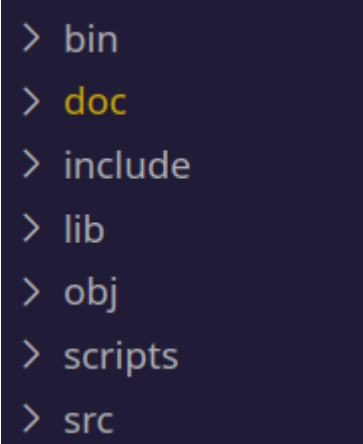
    files { "**.h", "**.c", "**.hpp", "**.cpp" }
    wx_config {Unicode="yes", Version="3.2", Libs="
        core,aui,gl"}
    filter "configurations:Debug"
        defines { "DEBUG" }
        symbols "On"

    filter "configurations:Release"
        defines { "NDEBUG" }
        optimize "On"
```

3 Structura Proiectului

3.1 Structura folderelor

Proiectul este structurat în mai multe foldere, fiecare cu un scop diferit descris mai jos. Mai jos se află o poză cu folderele proiectului:

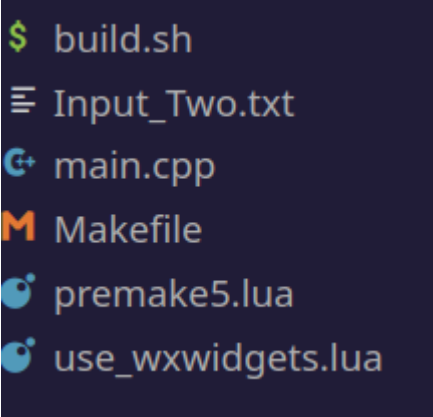


```
> bin
> doc
> include
> lib
> obj
> scripts
> src
```

Folderurile "doc", "obj" și "lib" nu sunt explicate, deoarece acestea conțin codul care generează documentația prezenta, obiecte generate în procesul compilării și fișiere pentru link dinamic nefolosit de proiect deoarece ele se află într-un folder extern.

3.2 Fișierele neclasificate

Fișierele care au un rol diferit de orice folder, astfel neputând fi conținute de niciunul. Aceste fișier sunt:



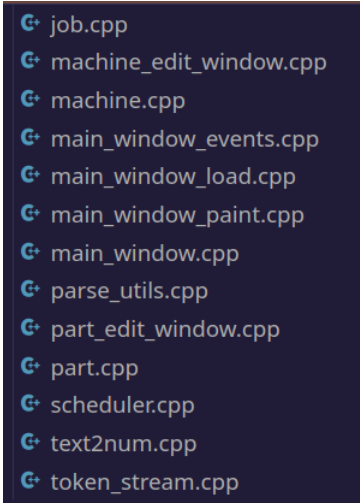
```
$ build.sh
≡ Input_Two.txt
G+ main.cpp
M Makefile
premake5.lua
use_wxwidgets.lua
```

Pe scurt, rolurile lor sunt:

- buildsh – un mic script bash pentru compilarea paralelizată a programului.
- Input_Two.txt – fișierul de intrare pe care este testat programul.
- main.cpp – fișierul principal care conține codul de inițializare grafică.
- Makefile – fișierul generat de build system pentru compilarea programului.
- premake5.lua – configurarea proiectului.
- use_wxwidgets.lua – script LUA pentru a include ușor librăria externă wxWidgets.

3.3 Folderul "src"

Folderul "src" conține toate definițiile claselor și funcțiilor proiectului, acestea sunt compilate în fișiere de tip ".o" care sunt ulterior link-uite împreună pentru a ajunge la executabilă finală. Fișierele din folderul "src":



```
job.cpp
machine_edit_window.cpp
machine.cpp
main_window_events.cpp
main_window_load.cpp
main_window_paint.cpp
main_window.cpp
parse_utils.cpp
part_edit_window.cpp
part.cpp
scheduler.cpp
text2num.cpp
token_stream.cpp
```

3.4 Folderul "include"

Folderul "include" conține toate declarările claselor și funcțiilor proiectului. Mai mult, conține și un fisier header a librăriei JSON for Modern C++, numit "json.hpp". Acest fișier este special, deoarece conține definiția și declararea tuturor claselor și funcțiilor din librărie, fără a fi nevoie de linking dinamic.

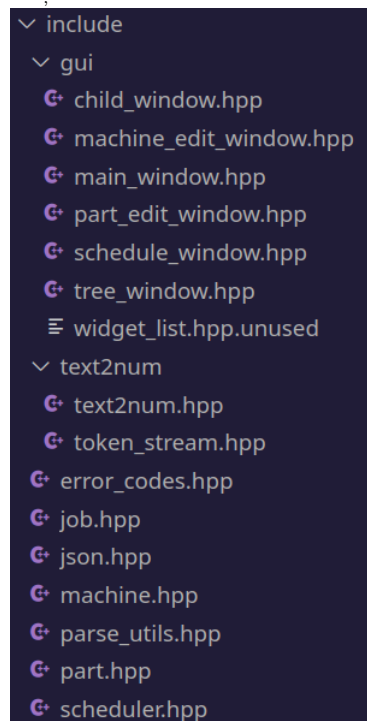
3.4.1 Subfolderul "gui"

Subfolderul "gui" conține declarări ale claselor legate de interfață grafică.

3.4.2 Subfolderul "text2num"

Subfolderul "text2num" conține declarările funcțiilor din librăria scrisă de mine numită "text2num", librărie care face conversia de la numere scrise în cuvinte, la valoarea lor corespunzătoare că număr scris în cifre.

Fișierele din folderul "include":



3.5 Folderul "bin"

Folderul "bin" conține executabilele proiectului adică rezultatele procesului de build. Este structurat în două subfoldere: "Debug" pentru executabilele cu simboluri de depanare și "Release" pentru executabilele compilate cu opțiunea de optimizare și fără simboluri de depanare.

3.6 Folderul "scripts"

Folderul "scripts" conține codul scris în Python, acest cod se ocupă cu vizualizarea calculelor făcute de programul principal C++. Motivul pentru care am ales acest stil de arhitectură este pentru a permite clienților să utilizeze propriile moduri de vizualizare a datelor.

4 Parsare

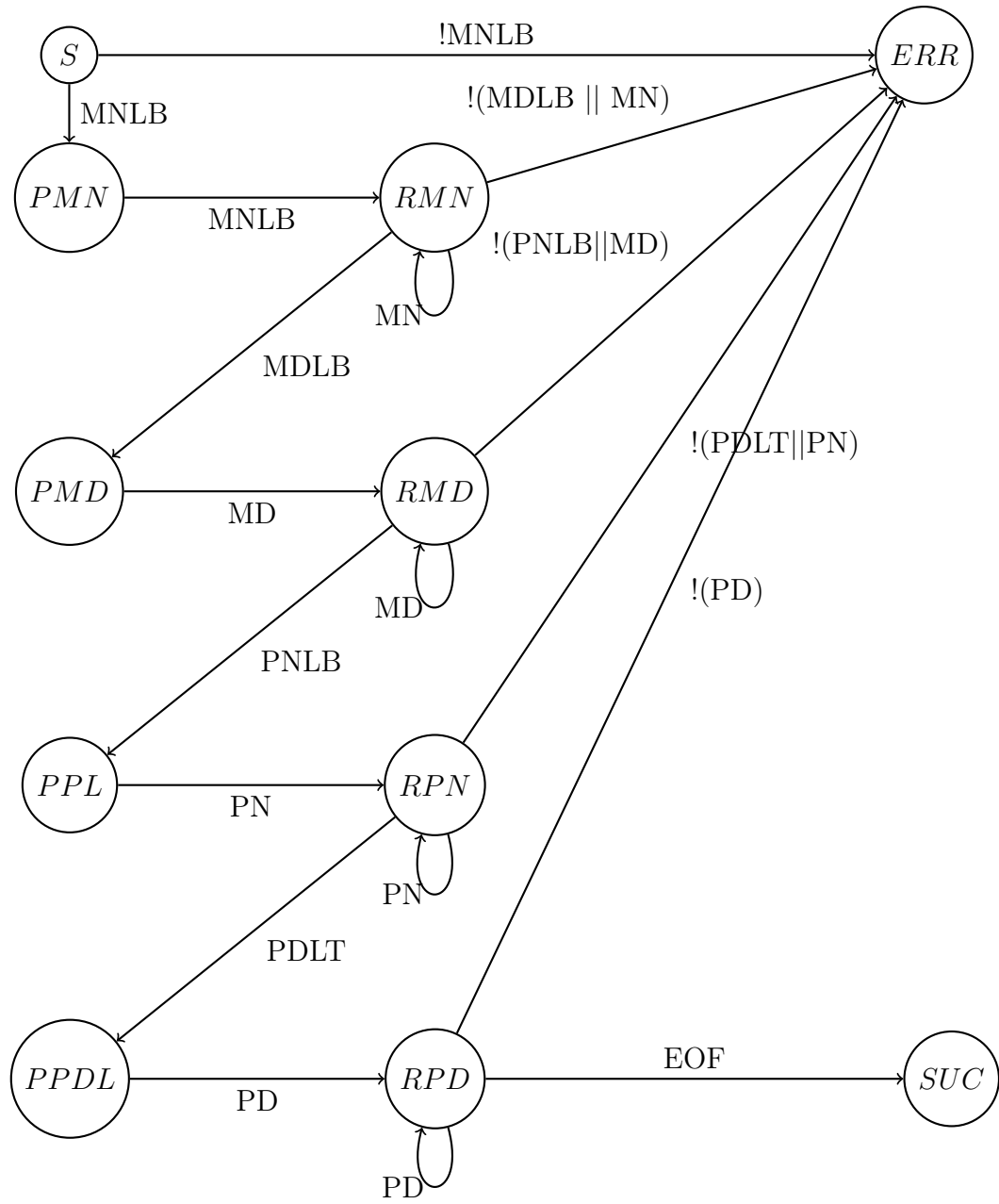
O bună parte a codului proiectului se ocupă de partea de parsare a fișierelor utilizatorului. Motivul pentru care această funcționalitate are așa mare importanță este pentru a facilita migrarea companiilor mici, fără infrastructură digitală, la această aplicație. Programul oferă două metode de parsare:

- Parsarea Fuzzy
- Parsarea JSON

4.1 Parsarea Fuzzy

Parsarea Fuzzy permite utilizatorului să facă greșeli în denumirile unei masinări, de exemplu, dacă acesta scrie "Fierăstrău Automat" prima dată iar mai apoi greșește și scrie "Ferăstrău Atutomat", programul va face legătură între cele două și va folosi formularea corectă. Totuși această metodă nu este corectă în proporție de 100% și poate da greși.

4.1.1 Mașină de stare



4.1.2 Legendă mașină de stadiu

Stări:

- S - Stare de start, verifică dacă e valid documentul.
- PMN – Stare care așteaptă simbolul pentru lista de nume și ID-uri userspace ale mașinărilor.
- RMN – Stare în care se parsează numele și ID-ul userspace al unei mașinării, până la simbolul pentru lista de descrieri ale mașinărilor.
- PMD – Stare care așteaptă și se verifică simbolul pentru lista de descrierile mașinărilor.
- RMD – Stare în care se parsează descrierile mașinărilor.
- PPL – Stare în care se așteaptă și verifică simbolul pentru lista de nume și ID-uri userspace ale părților care trebuie produse.
- RPN – Stare în care se parsează numele și ID-ul părților.
- PPDŁ – Stare în care se așteaptă și verifică simbolul care denotă îneperea listei de descrieri ale procesului de fabricare al unei părți.
- RPD – Stare în care se parsează descrierea procesului de fabricare al unei părți.
- ERR – Stare de ieșire, eroare la parsare.
- SUC – Stare de ieșire, document parsat cu succes.

Condiții de tranziție:

- MNLB – Simbol de început al listei numelor și ID-urilor userspace al mașinărilor.
- MN – Simbol pereche UserspaceID.Nume
- MDLB – Simbol de început al listei descrierilor mașinărilor.
- MD – Simbol care desemnează descrierea unei mașinării.

- PNLB – Simbol de început al listei numelor și ID-urilor userspace al părților.
- PN – Simbol pereche UserspaceID.NumeParte
- PDLT – Simbol de început al listei descrierilor părților.
- PD – Simbol care desemnează descrierea unei părți.
- EOF – Simbol care desemnează sfârșitul unui fișier.

4.1.3 Descriere detaliată

Fiecare stare din diagramă 4.1.1 are o implementare complexă și diferită. Arhitectură de parsare este una simplă și șablonată, urmărind un șablon liniar de parcurgere a stărilor.

4.1.3.1 Starea de început (S)

Starea de început va aștepta simbolul **availablemachines** pe care îl obține din începutul unui fișier de intrare fuzzy, adică **Available machines:**. Pentru a evita riscul unei parsări eșuate, verificarea utilizează o funcție care transformă simbolul scris de om, în unul ușor de verificat, exemplu fiind **Available machines:** \Rightarrow **availablemachines**. Codul funcției este următorul:

```
bool is_machine_list_begin(std::string& line){
    conv_to_parsable(line);
    return line.find("availablemachines") != std::
        string::npos;
}
```

```

void conv_to_parsable(std::string& line){
    std::string new_str;
    std::transform( line.begin(), line.end(),
                    line.begin(), tolower);

    for(auto & c : line){
        if(isalnum(c))
            new_str.push_back(c);
    }
    line = new_str;
}

```

4.1.3.2 Parsarea ID și Nume (PMN și RMN)

Parsarea ID și Nume se face după detectarea simbolului MNLB, prin citirea unei linii, extragerea primului număr până la întâlnirea unui caracter ne-numeric, extragerea apoi a primelor cuvinte până la întâlnirea unor caractere care nu aparțin alfabetului englez sau nu sunt **_** sau **-**. Codul verificării este următorul:

```

int parse_machine(std::string& line, Job* job){
    auto id = extract_first_num(line);
    auto m_name = extract_first_words(line, [](char c)
    ){
        return isspace(c) || isalpha(c) || c=='-' ||
            c=='_';
    });

    if(!has_chars(m_name)) return ERROR_BAD_NAME;

    while(isspace(m_name.back()))
        m_name.pop_back();

    for(auto & existing_machine : job->machines){
        if(id == existing_machine.get_id()){
            return ERROR_ID_EXISTS;
        }
    }
}

```

```
}  
  
Machine mach(m_name,id);  
job->machines.push_back(mach);  
return 1;  
}
```

4.1.3.3 Parsarea descrierilor mașinărilor(PMD și RMD)

Starea această este inițiată de simbolul **machinefeatures**.

Parsarea descrierilor mașinărilor este un procedeu delicat deoarece necesită că linia care specifică cooldown-ul să fie succesoare și direct sub linia care specifică capacitatea mașinăriei. Acesta este un lucru care, în viitor, poate fi îmbunătățit pentru a nu prezenta o astfel de contrangere majoră.

Comparativ cu parsarea de până acum, această stare citește câte două linii deodată și verifică dacă prima este cea de capacitate iar a doua de cooldown, emițând o eroare în cazul în care această cerință nu este îndeplinită. Parsarea unei caracteristici urmează următorul algoritm:

1. Din linia capacității, extrage ID-ul userspace al mașinăriei descrise.
2. Extrage de pe ambele linii textul care urmează ID-ul. începe după întâlnirea simbolului
3. Transformă în numere utilizabile de algoritm capacitatea respectiv cooldown-ul.

Ultimul pas al algoritmului este complex, parsarea fiind fuzzy, permite că numărul care definește capacitatea și cooldown-ul să fie scris cu cifre (1029) sau cu litere, în limba engleză (one thousand twenty nine), algoritmul folosindu-se de funcția **text2num** care poate parsă ambele cazuri în întregi pe 64 de biți.

Codul acestei stări este următorul:

```

int parse_machine_feature(std::string& line1, std::
string& line2, Job* job){
    auto machine_id = extract_first_num(line1);

    bool found_machine = false;
    for(auto & existing_machine : job->machines){
        if(machine_id == existing_machine.get_id()){
            found_machine = true;
        }
    }
    if(!found_machine) return ERROR_BAD_ID;

    std::string cap_str = extract_last_words(line1);
    std::string cool_str = extract_last_words(line2)
        ;

    std::size_t capacity = text2num(cap_str);
    std::size_t cooldown = text2num(cool_str);

    for(auto & existing_machine : job->machines){
        if(machine_id == existing_machine.get_id()){
            existing_machine.set_capacity(capacity);
            existing_machine.set_cooldown(cooldown);
        }
    }

    return 1;
}

```

4.1.3.4 Parsarea părților (PPL, RPN, PPDL și RPD)

Parsarea părților este implementată asemănător cu cea a mașinărilor, singură diferența fiind la parsarea descrierilor, unde pot exista un număr arbitrar de linii pentru o descriere, ceea ce a necesitat un algoritm mai bun, care consumă linii până detectează că a ajuns la o altă descriere moment în care se oprește din consumat și parsează descrierea.

4.1.4 Funcția `text2num`

Funcția **`text2num`** este o funcție care primește ca parametru un string care reprezintă în cuvinte a unui număr mai mic de 2^{60} și returnează un întreg cu semn de 64 de biți egal cu numărul reprezentat de string. Funcția nu are rată de succes 100% dacă numărul introdus nu urmează regulile de gramatică din limba engleză. Mai mult, pentru scopurile programului, funcția interpretează orice string care nu conține un număr ca 0 și ia ca prioritate cifrele din parametru înainte cuvintelor. De exemplu stringul "**120 five thousand forty five**" este 120 nu 5045.

Algoritmul din spatele funcției este următorul:

1. Verifică dacă există numere în parametru
2. Dacă există, formează un număr cu acele numere
3. Dacă nu există, interpretează parametrul.

Algoritmul de interpretare este următorul:

1. Sparge parametrul în tokenuri predefinite.
2. Transforma tokenurile în întregi.
3. Parcurge șirul de întregi.
4. Dacă următorul întreg este mai mic decât cel curent atunci adaugă la număr, altfel înmulțește.

De exemplu, stringul "**five thousand forty five**" devine șirul de întregi **5, 1000, 40, 5** care reprezintă următoarea ecuație:

$$5 \cdot 1000 + 40 + 5 = 5045$$

4.2 Parsarea JSON

Parsarea JSON facilitează siguranța datelor citite, deoarece în formatul JSON sunt salvate planurile de producție citite de Parsarea Fuzzy, după eventuala lor modificare în cazul erorilor.

Parsarea JSON este implementata folosind o librărie externa și nu necesita explicatii indetaliu. Formatul JSON este un format standardizat și folosit peste tot în industrie. Un exemplu mic de obiect JSON este:

```
{
  "nume": "Catalin Stefan Ion",
  "varsta": 47,
  "masina": null
}
```

5 Modelarea Datelor

Proiectul conține trei clase principale folosite pentru modelarea datelor. Acestea sunt **Job**, **Machine** și **Part**. Raționamentul din spatele acestei alegeri este structura fișierului de intrare și problema rezolvata de program. Structura fișierului este următoarea:

```
Available machines:
1. Band saw
2. Lathe
3. Knee Mill
4. Part washer
5. Dual-spindle machining center
6. Rotary tumbler

Machine features:
1:  - Capacity: one part at a time
    - Cooldown time: none

2:  - Capacity: one part at a time
    - Cooldown time: none

3:  - Capacity: one part at a time
    - Cooldown time: none

4:  - Capacity: one part at a time
    - Cooldown time: 600 seconds after each part
```

5: - Capacity: two parts at a time
- Cooldown time: none

6: - Capacity: no limit
- Cooldown time: none

Part list:

1. Door knob - 6 items
2. Pen - 12 items
3. Keyboard frame - 1 item
4. Intake manifold - 4 items

Part operations:

1: - Band saw: 150 seconds
- Lathe: 1200 seconds
- Part washer: 100 seconds

2: - Band saw: 50 seconds
- Lathe: 2000 seconds
- Knee Mill: 1200 seconds
- Rotary tumbler: 600 seconds
- Part washer: 200 seconds

3: - Band saw: 200 seconds
- Dual-spindle machining center: 8000 seconds
- Rotary tumbler: 600 seconds
- Parts washer: 600 seconds

4: - Band saw: 2000 seconds
- Knee mill: 4000 seconds
- Dual-spindle machining center: 3000 seconds
- Rotary tumber: 3500 seconds
- Part washer: 1200 seconds

Se observa că fișierul oferă informații despre mașinării și parti, adică clasele **Machine** și **Part**. Clasa de **Job** este clasa care reprezintă tot fișierul și care de unește cele doua clase mai mici.

5.1 Clasa Part

Clasa **Part** reprezintă o parte care trebuie produsa. Acestea are că membrii un ID și un nume împreună cu metode de tip getter și setter aferente, procesul de fabricare fiind retinut într-o structura de date în clasa **Job**.

Definiția clasei este:

```
#ifndef PART_HGUARD
#define PART_HGUARD
#include
#include
#include "machine.hpp"

class Part
{
private:
    std::string name;
    std::size_t id;
public:
    Part(std::string name) : name(name) {};
    Part(std::string name, std::size_t id) : name(
        name), id(id) {};
    Part(/* args */);

    std::string get_name();
    std::size_t get_id();

    void set_id(std::size_t new_id);
    void set_name(const std::string& new_name);

    ~Part();
};

typedef std::pair PartOrder;

#endif
```

Clasa definește și un tip de data numit **PartOrder** care reprezintă informația despre numărul de parti care trebuie fabricate.

5.2 Clasa Machine

Clasa **Machine** reprezintă o mașinărie din fabrică. Această are ca membrii un numele, un ID, capacitatea și cooldown-ul și metodele de get și set aferente.

Definiția clasei este:

```
#ifndef MACHINE_HGUARD
#define MACHINE_HGUARD
#include

class Machine
{
private:
    std::string name;
    std::size_t capacity;
    std::size_t cooldown;
    std::size_t id;

    bool in_use = false;
public:
    Machine(std::string name, std::size_t cap,
            std::size_t cooldown) : name(name),
                                   capacity(cap), cooldown(cooldown){};

    Machine(std::string name, std::size_t cap,
            std::size_t cooldown, std::size_t id)
        :
        name(name), capacity(cap), cooldown(
            cooldown),
        id(id){};

    Machine(std::string name, std::size_t id):
        name(name), id(id){};

    std::string get_name();
    std::size_t get_capacity();
    std::size_t get_cooldown();
    std::size_t get_id();
```

```

        void set_name(const std::string& new_name);
        void set_capacity(std::size_t new_cap);
        void set_cooldown(std::size_t new_cool);
        void set_id(std::size_t new_id);

        Machine(/* args */);
        ~Machine();
    };

#endif

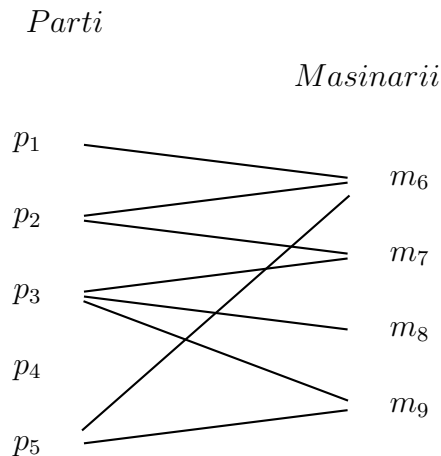
```

5.3 Clasa Job

Cea mai complexa dintre cele trei clase principale prin care datele sunt modelate este clasa **Job**. Aceasta are în compoziția ei trei structuri de date principale și un nume pentru a putea fi identificata

5.3.1 Structura de graf bipartit

Clasa folosește un graf bipartit pentru a reține procesul de fabricare a fiecărei parti.



Am ales acest mod de a reprezenta datele deoarece faciliteaza implementarea ușoară a algoritmului de generare a planului de producție, viteza de acces și economia de memorie.

5.3.2 Vectorii de parti și mașinării

Clasa mai cuprinde și doi vectori care rețin părțile respectiv mașinările descrise de fișierul de intrare. Am ales acest mod de a stoca mașinările deoarece faciliteaza căutarea după mai multe criterii sacrificând eficiența.

5.4 Constructorii

Clasa implementează un constructor default și unul care primește ca parametru un path spre un fișier de intrare unde vă folosi parsarea fuzzy pentru a creea un obiect din acel fișier. Mai mult, oferă funcționalitate de a se salva și de a se încarca din fișiere json.

Definiția clasei **Job** este;

```
#ifndef JOB_HGUARD
#define JOB_HGUARD
#include
#include
#include
#include

#include "part.hpp"
#include "machine.hpp"
#include "error_codes.hpp"
#include "text2num/text2num.hpp"
#include "parse_utils.hpp"

#include "json.hpp"
typedef std::multimap> bipartite_graph;

class Job
{
private:
public:
    bipartite_graph operations;
```

```

    std::vector orders;
    std::vector machines;

    std::string name;

    void add_operations(size_t id_part, size_t
        id_machine, float time);

    Machine& get_machine_by_id(std::size_t);
    PartOrder& get_order_by_id(std::size_t);

    std::optional<
    get_machine_by_name(const std::string&);

    std::optional<
    get_order_by_name(const std::string&);

    std::string as_string();

    void json_load(const std::string& path);
    void json_save(const std::string& path);

    Job(const std::string & path_to_config);
    Job(/* args */);
    ~Job();
};
#endif

```

6 Stocarea Datelor

Deoarece stocarea datelor este necesara, programul se folosește de formatul **JSON** pentru a stoca obiectele de tip **Job**. Deși citirea lor se poate face și din fișiere scrise și citibile de un om, stocarea se poate face doar în fișiere **JSON** pentru a asigura corectitudinea și viteza.

Funcția de salvare a unui obiect **Job** este:


```

void Job::json_save(const std::string& path){
    nlohmann::json j = {
        {"name",this->name}
    };

    j["machines"] = j.array();
    for(auto & machine : this->machines){
        nlohmann::json machine_j;
        machine_j["id"] = machine.get_id();
        machine_j["name"] = machine.get_name();
        machine_j["capacity"] = machine.get_capacity
            ();
        machine_j["cooldown"] = machine.get_cooldown
            ();
        j["machines"] += machine_j;
    }

    j["parts"] = j.array();
    for(auto & order : this->orders){
        auto process = this->operations.equal_range(
            order.first.get_id());
        nlohmann::json part_j;
        part_j["id"] = order.first.get_id();
        part_j["name"] = order.first.get_name();
        part_j["amount"] = order.second;
        part_j["process"] = j.array();
        for(auto op = process.first; op != process.
            second; ++op){
            nlohmann::json op_json;
            op_json["machine"] = op->second.first;
            op_json["duration"] = op->second.second;
            part_j["process"] += op_json;
        }
        j["parts"] += part_j;
    }

    std::ofstream out(path);

```

```
        out << std::setw(4) << j;
    }
```

Funcția de încărcare a unui obiect **Job** este:

```
void Job::json_load(const std::string& path){
    std::ifstream in(path);
    nlohmann::json j;
    j = j.parse(in);

    this->machines.clear();
    this->orders.clear();
    this->operations.clear();

    this->name = j["name"].get();

    for(auto& mj : j["machines"]){
        Machine machine(
            mj["name"].get(),
            mj["capacity"].get(),
            mj["cooldown"].get(),
            mj["id"].get());
        this->machines.push_back(machine);
    }

    for(auto& pj : j["parts"]){
        Part part(
            pj["name"].get(),
            pj["id"].get());

        this->orders.push_back({part, pj["amount"].
            get()});
        for(auto& opj : pj["process"]){
            this->operations.insert(
                {
                    part.get_id(),
                    {
                        opj["machine"].get(),
```

```

    opj["duration"].get()
    }
    }
    );
    }
    }
}

```

7 Interfață

Interfață proiectului este una simplă deoarece piață țintă este cea industrială, unde funcționalitatea este cheia și nu aspectul.

Aceasta este implementată folosind biblioteca **wxWidgets** care necesită implementarea unei anumite arhitecturi. La pornire programul lansează un obiect de tip **App** care conține un obiect de tip **Window**, unde funcționalitatea aplicației este implementată prin răspunsuri la evenimente cum ar fi click-ul, apăsarea unei taste, resizing etc.

Definiția aplicației principale este:

```

#ifndef MAIN_WINDOW_HGUARD
#define MAIN_WINDOW_HGUARD

#define DEBUG_PRINTS

#include
#include
#include
#include
#include
#include

#include
#include
#include
#include

#include "../job.hpp"

```

```

#include "part_edit_window.hpp"
#include "machine_edit_window.hpp"
#include "../scheduler.hpp"

class App : public wxApp
{
public:
    bool OnInit() override;
};

class MainWindow : public wxFrame
{
public:
    MainWindow();

private:
    std::unordered_map all_jobs;
    ChildWindow* edit_window = 0;

    wxTreeCtrl* tree;
    wxTreeItemId jobs_root;
    wxTreeItemId active_item;
    wxBoxSizer* vbox;
    Job* job = nullptr;

    void OnHello(wxCommandEvent& event);
    void OnExit(wxCommandEvent& event);
    void OnAbout(wxCommandEvent& event);

    void OnLoad(wxCommandEvent& event);
    void OnLoadJSON(wxCommandEvent& event);
    void OnSaveJSON(wxCommandEvent& event);
    void OnCalculate(wxCommandEvent& event);

    void ActivateMainWindowTools();
    void BuildWindowLayout();

```

```

void OnSelectedTreeItem(wxTreeEvent& event);
void OnPaintTree(wxPaintEvent& event);

void PartNameCallback(wxCommandEvent& event);
void PartAmountCallback(wxCommandEvent& event);

void MachineNameCallback(wxCommandEvent& event);
void MachineCooldownCallback(wxCommandEvent&
    event);
void MachineCapacityCallback(wxCommandEvent&
    event);

void RemoveAllChildWindows();

wxDECLARE_EVENT_TABLE();

};

enum
{
    ID_Hello = 1,
    ID_LoadJob = 2,
    ID_Tree = 4,
    TEST_TextBox = 3,
    ID_LoadJSON = 5,
    ID_SaveJSON = 6,
    ID_Calculate = 7
};

#endif

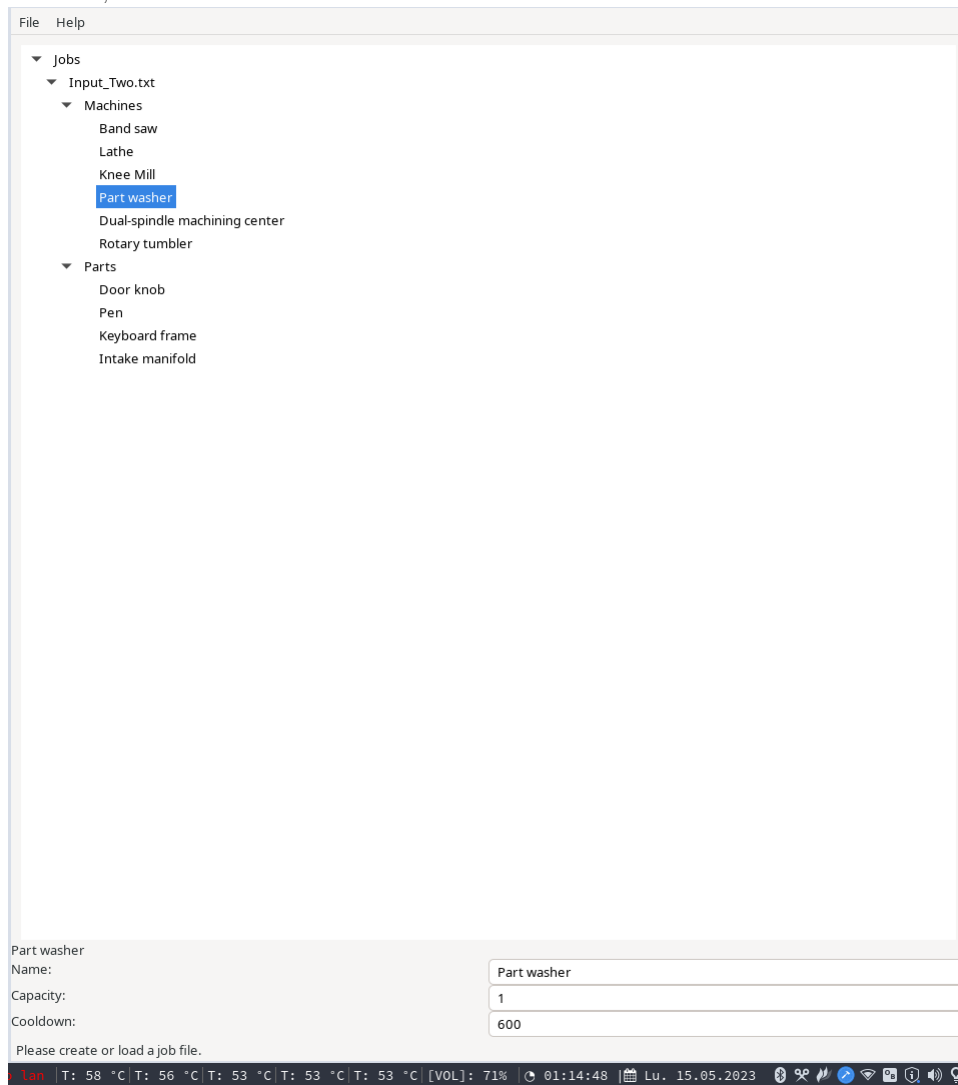
```

7.1 Funcționalitate

Interfață oferă posibilitatea de a modifica caracteristicile unei mașinării dar și a unei parti. Mai mult aceasta permite încărcarea, salvarea și generarea planului de producție printr-un **menu bar** sau prin scurtături de la tastatură.

7.2 Poze

Interfață:



8 Descrierea Problemei

Problema este de a proiecta un software care programează operațiunile pe piese în cel mai eficient mod posibil. Eficiență, în acest context, se măsoară prin cât de repede este rulat un set de piese prin toate operațiunile prevăzute.

9 Îmbunătățiri pe viitor

9.1 Parsare

La capitolul parasangare fuzzy o valoroasa îmbunătățire ar consta în eliminarea constrângerii că linia de capacitate și cea de cooldown să aibă o ordine și să fie una după alta, deoarece acest lucru este predispus la erori umane.

9.2 Programarea operațiunilor

Deoarece problema propusa nu este încă rezolvata și nu exista un algoritm polinomial pentru toate cazurile și toate constrângerile, singura îmbunătățire este o aproximare mai buna a programului de fabricare.

9.3 Vizualizare

În momentul de fata, aplicația emite un set de numere care trebuie ulterior interpretat de utilizator în modul în care își dorește. O eventuala imbunatatire este să existe funcționalitate pentru vizualizare în moduri predefinite.

10 Generarea planului de producție

Algoritmul de aproximare a planului optim de producție consta în mai mulți subalgoritmi:

```
build_id_map();  
create_dataframe();  
build_time_matrix();  
build_machine_queue();  
build_insertion_queue();  
dfs_intersection_sort();
```

10.1 Construirea IDMap-ului

Structura de date IDMap consta într-o structura de tip map care servește în convertirea din ID-urile userspace ale mașinărilor și părților în ID-uri Algo-

space, adică id-uri uniforme care încep de la 1 și se termina la N (numărul de mașinării).

10.2 Graful de producție

Graful de producție este similar cu un graf orientat normal dar consta în doua tipuri de noduri diferite.

10.2.1 Nodul Parte

Imaginând o pagina goala ipotetica, nodurile parte sunt reprezentate în linie cel mai sus în pagina.

10.2.2 Nodul Mașină

Pe aceeași foaie, pentru fiecare ID Algospace se vă desena orizontal un rand de mașinării cu acel ID în număr egal cu capacitatea mașinării cu ID-ul respectiv.

10.2.3 Procesul de legare

De la fiecare nod parte, se vă crea o muchie orientata înspre mașinăria care urmează în pasul de producție următor. Acolo unde deja exista o muchie, se vă adaugă încă una, existând posibilitatea că între doua noduri mașină să exista M , $M \leq N$ miuchii.

10.3 Construirea Dataframe-ului

Structura de dataframe reprezintă o matrice tri-dimensională. Aceasta matrice servește că și o matrice de adiacenta speciala, deoarece este necesara reprezentarea drumului fiecărei parti prin **graful de producție** atunci când exista posibilitatea intersectării a doua drumuri deoarece doua mașinării au procese de fabricare asemănătoare.

10.4 Construirea Time Matrix-ului

Structura de time matrix este o matrice normala cu numere reale, de dimensiuni $M \times N$ unde M este numărul de parti și N numărul de mașinării.

Forma generala a aceste matrici este:

$$\begin{bmatrix} p_{11} + c_1 & p_{12} + c_2 & \dots & p_{1n} + c_n \\ p_{21} + c_1 & \dots & \dots & p_{2n} + c_n \\ \dots & \dots & \dots & \dots \\ p_{m1} + c_1 & \dots & \dots & p_{mn} + c_n \end{bmatrix}$$

Unde p_{mn} este timpul de fabricare a părții cu ID m în mașinăria cu ID n iar c_n este cooldown-ul mașinăriei cu ID n . Aceasta matrice permite eliminarea constrângerii cooldown-ului mașinăriilor.

10.5 Construirea cozilor și sortarea

Algoritmul folosește doua cozi. Coadă **MachineQueue** reprezintă ordinea în care fiecare mașinărie procesează părțile. Coadă **IntersectionQueue** este o coada sortata care reprezintă pentru fiecare mașinărie, partea cu cel mai scurt timp și care începe procesul de producție pe mașinăria respectiva.

Sortarea **DFS Intersection Sort** este procedeul prin care algoritmul aproximează planul de producție, acesta adăugând în coada fiecărei mașinării partea cu cel mai scurt timp inițial de producție, apoi executând un DFS pentru a adaugă partea aceea și în restul mașinăriilor care fac parte din procesul de fabricare. Eventualele parti adăugate ulterior vor fi adăugate astfel încât să fie minimizat timpul mort pe fiecare mașinărie.

11 Cerințe Software și Hardware

Cerințele necesare fiecărei librării folosite.