Readme Tema 1 IA

Detalii implementare:

util.py

- class Task - contine campuri caracteristice unui obiect de tip task (index, durata, deadline, lista de task-uri anterioare, timpul la care a fost planificat si procesorul pe care a fost planificat). In plus, exista metode pentru calcularea timpului de terminare a task-ului (getFinishTime) si pentru determinarea costului pe care il adauga acest task la solutia problemei (getCost).
- class State - defineste un state al problemei. State-ul este determinat de configuratia actuala a procesoarelor (ce task-uri au fost planificate si pe ce procesor) si ce task-uri au ramas de planificat. Procesoarele sunt modelate simplu, ca niste liste in care se vor stoca indecsii task-urilor ce vor fi planificate pe ele, in ordine. Pentru a putea identifica task-urile, se pastreaza un dictionar cu toate task-urile, iar pentru a nu fi nevoie sa se itereze prin el, task-urile neplanificate sunt stocate ca o lista de indecsi. O stare este considerata finala daca nu mai ramane nicio sarcina de planificat. Exista functii pentru verificarea validitatii unei stari partiale (pentru fiecare task se itereaza prin lista de dependinte si in cazul in care acestea au deja planificate, ne asiguram ca ele se termina inainte ca sarcina curenta sa inceapa; in cazul in care nu au fost planificate, le ignoram (luand in considerare faptul ca ele ar putea sa fie planificate mai tarziu in functia de cautare, insa mai devreme temporal pe unul dintre procesoarele mai libere)

search.py

- functia main in care se parseaza argumentele primite si se apeleaza functia de cautare
- functia de cautare care are diverse argumente: poate sorta sarcinile in functie de deadline, le poate sorta topologic, poate sorta procesoarele, poate face orice combinatie dintre acestea.

Functia de cautare primeste ca parametru un state. Daca este final si reprezinta o solutie vallida, Daca da, se updateaza cea mai buna solutie. Daca nu, se incearca planificarea fiecarui task pe fiecare procesor, se verifica daca ar fi o stare valida si se merge mai departe in recursivitate.

Euristici/Metode folosite:

1. **Sortarea task-urilor in functie de deadline**

In ciuda intuitiei din spatele rationamentului de a sorta task-urile in functie de deadline-ul lor, oferind prioritate celor care ar trebui sa fie terminate mai devreme, metoda nu gaseste solutii avand un buget mai mic, ci de cele mai multe ori nu gaseste deloc sau, atunci cand gaseste, ele sunt mai bune decat solutiile gasite cu backtracking-ul simplu.

2. **Sortarea procesoarelor in functie de load**

Daca procesoarele nu sunt sortate in functie de load, functia de cautare incearca de fiecare data sa planifice primul task neplanificat pe primul procesor, apoi, recursiv, acelasi lucru, obtinand in final solutii care nu sunt deloc echilibrate (de exemplu, toate sarcinile planificate pe un singur procesor si restul procesoarelor complet neutilizate). Atunci cand le sortam, ne asiguram ca de fiecare data mentinem un oarecare balans intre load-urile tuturor procesoarelor si implicit scadem costul solutiilor gasite (deoarece unele task-uri se rezolva in mod paralel si datorita lor sarcinile care se rezolva catre final vor avea o depasire a deadline-ului mai mica)

3. **Arc consistenta**

Avand in vedere ca un task poate fi planificat in functia de cautare inainte ca toate dependintele sale sa fi fost planificate (sperand ca acestea vor putea fi planificate mai tarziu in functia de cautare, insa respectand constrangerile), o forma de restrangere a spatiului de cautare este sa verificam daca o sarcina anterioara care nu a fost deja planificata, poate fi planificata ulterior in functia de cautare astfel incat ea sa se termine inainte sarcinii curente. Facand aceasta verificare avem certitudinea ca in viitor acea configuratie nu va duce la un dead-end. Cu toate acestea, aceasta forma de arc consistenta nu are niciun efect asupra backtracking-ului simplu (in primele cateva mii sau chiar milioane de iteratii), intrucat el planifica intotdeauna sarcinile initial pe primul procesor, iar celeltalte, fiind libere, garanteaza intotdeauna ca o dependinta care nu a fost deja planificata va putea fi planificata la timpul 0. Exista totusi situatii cand aceasta consistenta are efecte, cum ar fi: la testul 5, sortarea task-urilor dupa deadline impreuna cu sortarea procesoarelor nu duc la nicio solutie. Cand adaugam si conditia de arc consistenta, insa, se gaseste o solutie.

4. **Sortarea sarcinilor in functie de nivelul pe care l-ar ocupa intr-o sortare topologica, iar in caz de egalitate, sortarea dupa cel mai mic deadline “accesibil” (catre care exista cale din nodul curent) si dupa durata task-ului**

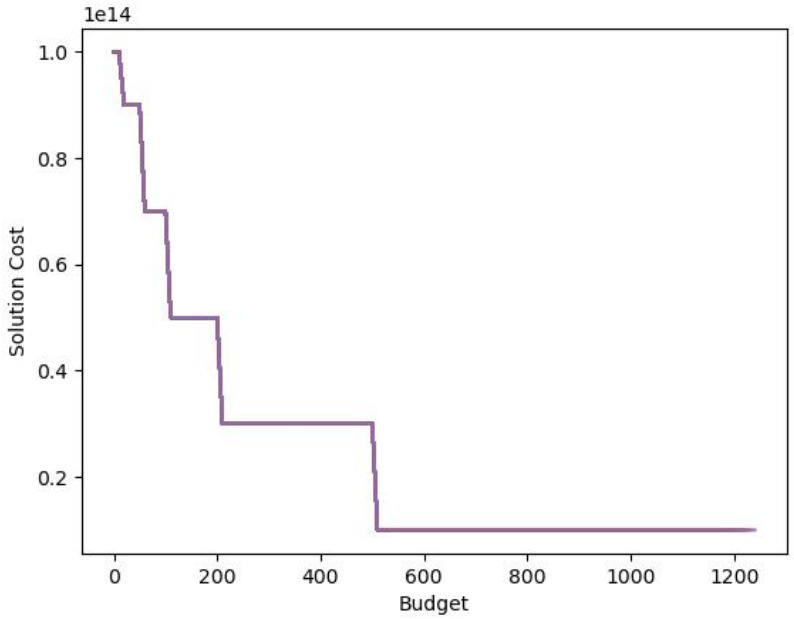
Pentru a ne asigura intotdeauna ca toate constrangerile referitoare la ordine sunt indeplinite, in functia de cautare vom ordona task-urile in ordine topologica, iar apoi fiecarui nod ii va fi asignat un scor (nodurilor fara in-muchii li se va da scorul 0), iar urmatoarelor noduri li se va asigna un scor folosind: scor[nod] = max(scor(in-nod)) + 1 (adica scorul unui nod este nivelul pe care acesta se afla). Avand in vedere ca o multime de task-uri pot avea scor 0 la un moment dat (toate au dependintele deja planificate), este nevoie de un criteriu de departajare, iar prioritate au sarcinile care au un deadline_scor cat mai mic (unde deadline-scor este deadline-ul minim al unui nod catre care exista cale din nodul curent), iar apoi sarcinile care au cea mai mica durata. Sortarea topologica reprezinta o forma de cale consistenta, in sensul ca toate constrangerile vor fi indeplinite oricum s-ar alege valori in cadrul asignarii curente (Daca tocmai am asignat un timp pe un procesor pentru task-ul T_i, oricare ar fi un task T_j inca neasignat, pentru orice valoare valida care ar putea fi asignata lui T_j, stim sigur ca oricare ar fi alta sarcina T_k care trebuie planificata, va putea fi automat planificata ulterior, deoarece ea este pe un nivel ulterior)

Rezultate obtinute:

1. **Evolutia mediei costului tuturor solutiilor gasite pentru problemele 1 - 10 in functie de bugetul acordat (Fig. 1)**

Initial, costul total este ~1e14, deoarece se porneste de la buget 0 (caz in care nu se gaseste nicio solutie, iar costul considerat este INF=99999999999999). Fiecare descrestere abrupta a graficului semnifica descoperirea unei solutii pentru cel putin o problema. Se observa ca dupa aproximativ 500 de iteratii, metoda backtracking nu reuseste sa gaseasca nicio solutie mai buna, iar in plus, pentru unele probleme (de exemplu p10), nu gaseste solutie deloc.

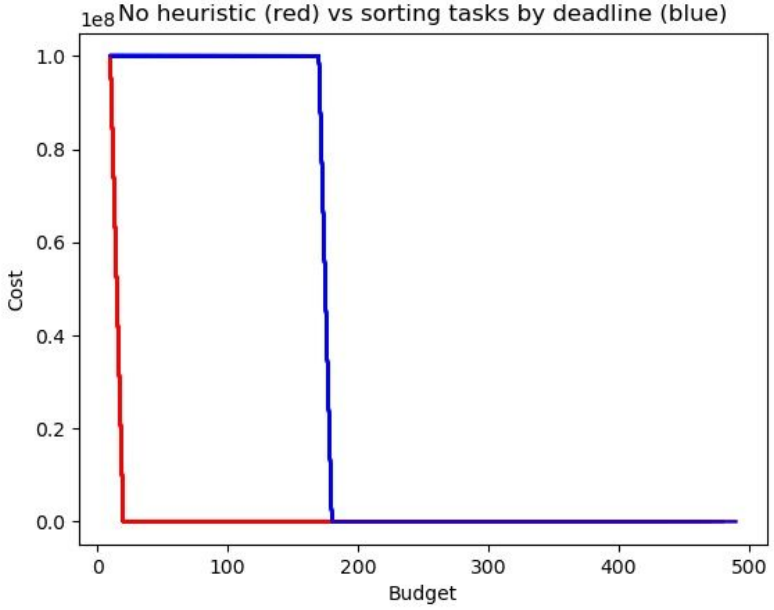
Fig. 1 - evolutie backtracking (media costurilor solutiilor tuturor problemelor)



2. Comparatie intre backtracking simplu si backtracking combinat cu sortarea task-urilor in functie de deadline (iar in caz de egalitate, sortarea crescatoare in functie de numarul de dependente nerezolvate, oferind astfel prioritate task-urilor care nu mai au task-uri anterioare ce ar trebui rezolvate) (Fig. 2)

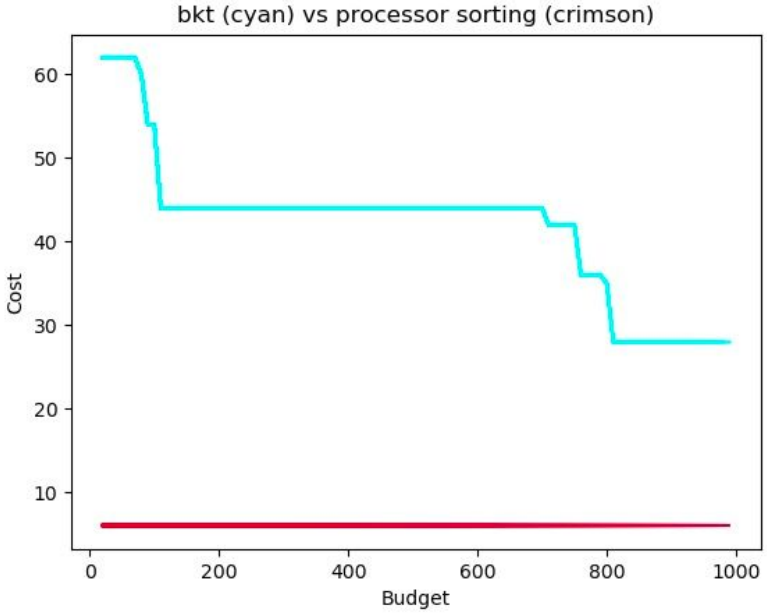
Desi metoda backtracking simpla gaseste o solutie mult mai repede (de la buget 20), pe care apoi o imbunatateste dupa un numar de pasi (de la cost 62, la 54, apoi 44), metoda care foloseste sortarea task-urilor dupa deadline gaseste prima solutie mai greu (dupa 180 iteratii), insa are un cost mult mai mic, de doar 22, iar ulterior gaseste solutii de cost 16 (la aproape 500 iteratii).

Fig. 2 - backtracking vs task sorting (problem 1)



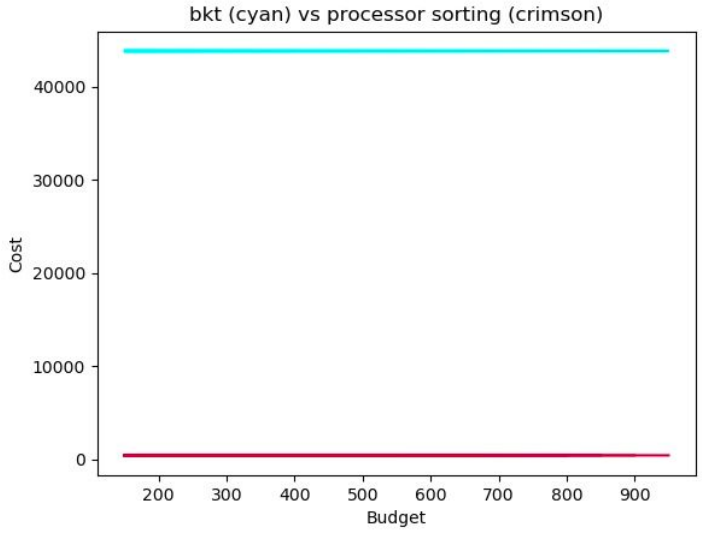
3. Comparatie intre backtracking fara euristici si sortarea procesoarelor (astfel incat primul procesor vazut este cel care are primul timp valabil cel mai mic) (Fig. 3)
Sortarea procesoarelor face posibila gasirea unei solutii foarte bune (cost 6 in doar 20 de iteratii), ulterior aceasta nu pare sa mai evolueze si sa gaseasca solutii mai bune. Fara nicio euristica, in schimb, pana la bugetul maxim 1000 se gasesc diverse solutii cu cost descrescator, insa niciuna la fel de buna ca cea gasita atunci cand procesoarele sunt sortate.

Fig. 3 - backtracking vs sortarea procesoarelor (problema 1)



Desi in cazul curent diferenta de cost nu este foarte mare, exista cazuri in care costurile celor 2 solutii sunt de mii sau zeci de mii de ori mai mari, asa cum se intampla in cazul problemei 4, unde folosind un buget de 150, folosind backtracking simplu gasim o solutie de cost 43786, iar folosind procesoarele sortate obtinem o solutie de cost 426, iar pana la 1000 de iteratii niciuna dintre acestea nu mai evolueaza. (Fig. 4)

Fig. 4 - backtracking vs sortarea procesoarelor (problema 4)



4. Backtracking simplu vs arc consistenta + sortarea procesoarelor + sortare task-uri dupa deadline vs arc consistenta + sortarea procesoarelor + sortare topologica task-uri (Fig. 5, Fig. 6)

Dupa cum se observa, combinarea mai multor euristici ajuta la gasirea mai rapida a unei solutii mai eficiente.

Fig. 5 (date obtinute pe problema 1)

Fig. 6 (costul reprezinta media costurilor tuturor solutiilor pentru p1-p10)