SELEÇÃO DE TAREFAS COM PONDERAÇÃO

Algoritmos e Estruturas de Dados (40437)

2020/2021

universidade de aveiro

Trabalho realizado por:
Tiago Alvim 95584 49%
Vasco Costa 97746 51%

Introdução

O problema apresentado consiste na escolha de determinadas tarefas que dê o maior lucro ou no caso destes serem iguais, o maior número de tarefas a serem realizadas. Sendo possível também fazer a simulação da variação do número de funcionários. Este é um problema relevante uma vez que permite a um empregador fazer o estudo de qual o melhor número de empregados a manter, daí a alteração na função de geração dos números aleatórios para não depender do número de programadores. Esta classe de problemas é bastante interessante na elaboração de algoritmos pois é um bom desafio assim como apresenta soluções muito práticas e utilizáveis no mudo real.

Abordagem computacional

1ª Abordagem (*dumb_approach*)

A abordagem inicial consistia em gerar-se recursivamente todas as permutações binárias, de modo a terem-se todas as hipóteses, num *array*. Posteriormente é percorrido esse array, indicando 0 e 1 para cada um dos índices, que são as tarefas, tendo o 0 o significado de não ser realizada e o 1 ser feita. Se a tarefa for para realizar então é necessário ver se existe algum programador para a desempenhar, se sim então esse é atribuído para a mesma, não havendo então essa hipótese não é válida e passa-se para a seguinte combinação binária.

Esta abordagem apresentou como desvantagem a computação de elementos que já tinham sido verificados como não sendo possíveis. A título de exemplo, imaginemos 4 tarefas, apenas é possível realizar a primeira, a segunda já não o é, com este método teríamos que analisar, considerando a segunda tarefa a contar se a terceira era possível ou não e ainda depois, se a quarta poderia ser feita ou não para as restantes combinações. Ou seja, é bastante ineficiente. Alertados por isto tomámos uma segunda abordagem.

2ª Abordagem (*generate_possibilities*)

Esta opção, sendo a que se optou por usar para a obtenção de resultados, passa por inicialmente colocar todas as hipóteses como não estando atribuídas a ninguém. E a criação de um *array* com o tamanho de todos os lucros possíveis mais um, para que a cada índice se possa dizer que aquele lucro ocorreu x vezes. Esta criação é simples, sendo que ocupará mais memória o que poderia ser evitado com dimensão dinâmica ao custo de algum desempenho, no entanto, optou-se por eficiência computacional com maior uso de memória.

Seguidamente tomou-se a abordagem de resolver o problema localmente, passa-se a tarefa atual à frente chamando a função recursivamente. Consequentemente considera-se a tarefa atual, verificando se existe algum programador livre, analisando se a data de início da tarefa que se está a considerar é maior que a data até à qual ele está ocupado. Havendo um programador livre então este passa a ter uma nova data até à qual está ocupado e a tarefa passa a estar atribuída, assim como o lucro total desta combinação aumenta com a realização da mesma, não sendo necessário ver se mais algum está livre. Na eventualidade de não existir nenhum programador livre, então a tarefa não é atribuída a ninguém. Nesse caso é escusado analisar o resto das situações que considerem esta tarefa como possível. E passa-se então a chamar a função com a tarefa seguinte.

Chegando ao fim, ao número de tarefas total, verificamos que se se chegou aí então é porque estamos perante uma combinação possível de tarefas. Caso o lucro que este conjunto

apresente seja superior ao que se obteve com outra combinação, então é a escolhida como melhor.

Regressando ao exemplo anterior com 4 tarefas possíveis. Utilizando a primeira abordagem, teríamos que considerar 2^4 casos, e depois ainda utilizar um método de n^2 para percorrer todas as tarefas e os programadores. Aqui, apesar de se continuar com uma complexidade computacional semelhante, as constantes escondidas pela notação assintótica fazem um grande efeito em termos de desempenho.

Importa salientar que falar de métodos de n^2 nestas situações é importante, pois não estamos a lidar com números do tamanho do universo e por isso a notação assintótica não conta toda a situação sendo as constantes que esta esconde bastante relevantes.

Situação de um programador e sem os lucros importarem

Esta situação é bastante restrita, no entanto, a vantagem que apresenta é bastante grande e de fácil implementação, valendo assim a pena o seu uso. Nesta situação o mais pertinente é o número de tarefas que é possível realizar e por isso a ordenação das tarefas pela data de fim é o ideal, percorrendo as tarefas assim organizadas e verificando se a adição dessa tarefa é compatível, isto é, se o programador está livre. Prova-se que a ordenação pelo fim da tarefa é preferível uma vez que suponhamos uma tarefa com início em *a* e fim em *b*, e duas tarefas, uma com início em *a* e fim em *a/2*, e outra com início em *a/2* e fim em *b*, com este método, verifica-se que as duas tarefas têm prioridade em relação à "maior" pelo que, permite a realização de mais tarefas.

Gen2

No seguimento da ideia apresentada no ponto anterior tivemos a ideia de que se funciona para um programador, então para mais que um talvez fosse um bom método a utilizar. No entanto, ao contrário do caso anterior, não existe uma prova de que é um método certo e como podemos verificar, este método não dá os melhores resultados.

Método da sorte (*random_approach*)

Por vezes existem problemas computacionais que devido à dificuldade de busca de um método analítico são impossíveis, não existindo solução. Neste caso existe forma analítica, mas também é possível tomar-se outra abordagem. Nomeadamente na função *random_approach* fazem-se um milhão e duzentas mil tentativas com combinações válidas. No início de cada uma, atribuem-se as tarefas a ninguém assim como colocamos os programadores livres. Após isto percorremos cada uma das tarefas. E usando *rand()* ou se considera ou não se considera a tarefa atual, caso se considere então é necessário verificar se existe algum programador livre como se fez no método analítico. Chegando à última tarefa, compara-se o lucro que esta hipótese teve com os casos anteriores, e na eventualidade de ser superior então esta é a hipótese que guardamos. Ao fim do milhão e duzentas mil combinações possíveis, não conseguimos dizer que obtivemos a melhor combinação. Uma vez que apenas podemos concluir que não fizemos todas as hipóteses, por exemplo, na situação de 34 tarefas e 2 programadores, para o NMec. 97746 existiam 104930192 combinações válidas, ou seja, um número superior a 1200000. Existindo sempre a hipótese de se repetirem combinações pois não se faz essa verificação. E esse é o problema deste método, apenas temos uma boa solução, não a melhor.

Análise[1]

Vamos agora ver e comparar os dados da nossa abordagem recursiva que utiliza a programação dinâmica (*generate_possibilities*). Que foi o método de onde retirámos os resultados. Apresentando primeiramente a tabela dos lucros obtidos:

97746	Programadores							
Tarefas	1	2	3	4	5	6	7	8
1	3102	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**
2	3618	3618	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**
3	2613	5211	6380	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**
4	3838	6365	8285	9290	**NULL**	**NULL**	**NULL**	**NULL**
5	6652	11041	13313	14815	14815	**NULL**	**NULL**	**NULL**
6	6756	9112	12004	13117	14020	14020	**NULL**	**NULL**
7	7114	5168	6264	8090	9782	11338	11878	**NULL**
8	12202	8037	10129	11648	13154	14419	15151	15151
9	10166	12120	14179	17778	21159	22156	23025	23773
10	11433	8955	12482	12631	14894	16446	17770	19065
11	14057	13839	14330	13953	16219	17957	19640	21274
12	9984	16475	14890	14781	17291	19681	22043	23094
13	21136	16252	15404	18816	19673	21871	23293	24307
14	15539	17040	16630	19545	17022	18770	20062	21347
15	23243	27566	22579	24621	18185	20091	21943	23666
16	18931	25477	24648	22609	18629	23634	26127	28116
17	19053	18448	22191	23907	20873	24416	26759	28916
18	23552	18117	20894	20199	18019	21467	22855	24109
19	18315	22621	26356	19643	21632	25757	27062	29392
20	26165	23839	23857	34223	30744	30612	31854	34875
21	37169	24070	30510	24851	24310	29067	26818	29081
22	38111	39681	25675	28539	28174	26533	28791	28066
23	37575	24317	38314	26887	29243	33382	31756	35555
24	29279	24559	28555	25211	24916	25652	27508	24114
25	41083	37558	33957	37830	35646	31170	32924	37227
26	19323	25348	27523	29018	32301	27515	31421	27692
27	30329	29750	43507	36647	31744	29627	34201	32214
28	38321	36092	36592	28281	32728	37497	37926	33610
29	40780	56525	57893	47964	40782	55505	42887	47018
30	46876	41112	52484	35295	37852	37805	39802	45812
31	29124	55513	37967	42488	45285	47208	42254	44972
32	55597	48179	48921	61305	45206	49768	43173	42325
33	49379	41945	50881	49837	45267	43582	47562	38737
34	40699	46736	50693	59719	49113	50663	51971	49145
35	46322	63133	46782	58071	50040	52412	48779	45719
36	45223	66111	45778	51110	54764	53748	52252	50245
37	60689	55213	51315	78154	56856	56918	59983	52925
38	55649	59940	55108	50471	70667	52750	54353	56570
39	56562	66206	53553	64337	62403	55486	55992	62373
40	70898	67368	68417	54216	55224	62165	60430	51142

[1] Os tempos computacionais e resultados foram obtidos com um processador *Ryzen 7 3700X* com uma velocidade base de 3,59 GHz. E 16 GB de memória *RAM* a 3333 MHz com CL16 em *dual-channel*.

95584	Programadores							
Tarefas	1	2	3	4	5	6	7	8
1	1550	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**
2	2671	4576	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**
3	2700	4557	5726	**NULL**	**NULL**	**NULL**	**NULL**	**NULL**
4	3035	5807	7296	8129	**NULL**	**NULL**	**NULL**	**NULL**
5	5991	7029	7791	7791	7791	**NULL**	**NULL**	**NULL**
6	5076	5531	6896	7799	8309	8309	**NULL**	**NULL**
7	7184	7644	10462	12107	13715	15048	15048	**NULL**
8	10298	11373	9596	11178	11853	11853	11853	11853
9	10945	10572	14900	18639	21350	22363	23174	23174
10	6783	11408	11998	13959	15558	16833	16833	16833
11	11544	9692	11223	11510	13680	15694	17410	18909
12	9956	16509	12932	12677	14844	16417	17780	19060
13	17080	12531	15239	13213	17640	20076	22418	24593
14	13088	14439	13940	17249	18689	20694	22594	24073
15	11322	15382	17772	13694	15963	17951	19892	21790
16	28794	21672	21483	23656	18310	23549	25815	27779
17	22887	20389	23688	25808	19915	24465	27178	29859
18	29331	34435	25255	23487	28370	26928	29508	31719
19	29031	22086	26009	25458	24931	23783	24840	27488
20	24556	25587	31199	25287	27962	29260	24998	26369
21	23162	30585	28949	28294	27221	24298	24631	26774
22	19410	27607	25352	35974	28082	28499	33577	32304
23	40017	40654	27814	30580	29098	31244	28664	31930
24	29265	44418	30202	38178	37770	35816	32142	37844
25	33758	35167	37314	28453	37812	26843	31278	35847
26	27543	33666	33740	36740	32796	34974	28945	32871
27	29749	33750	47960	35138	33112	32103	39589	39596
28	53699	31035	35919	42437	40353	41430	35629	33074
29	36085	53774	45691	47359	48280	48621	44701	43424
30	37400	41682	49447	48379	48090	48489	48970	44983
31	42411	49327	41453	49081	40775	43568	40381	44385
32	45237	42829	45767	44977	37656	46542	36500	38049
33	52734	39654	48186	49359	51042	50703	49571	49157
34	39758	41524	44342	42252	37790	44528	44861	41144
35	55805	54252	44275	53112	53514	52007	55111	47296
36	38897	45423	61798	42033	55959	55025	49036	44905
37	43275	42085	52212	59568	64885	44849	49852	49808
38	54423	81628	48125	58407	50445	55654	54770	53320
39	45000	47524	59041	55383	63072	58753	53860	56854
40	56838	57642	54806	55013	56397	58739	57627	57592

Pela leitura das tabelas concluímos que entre os dados retirados de cada um dos números mecanográficos os lucros não variam em grande escala, normalmente aumentando com o número de tarefas e com o número de programadores, embora não se verifique sempre.

Focando agora nos tempos do programa decidimos fazer um gráfico 3D, (um para cada aluno), onde mostramos o tempo numa escala logarítmica em função do número de programadores e do número de tarefas.

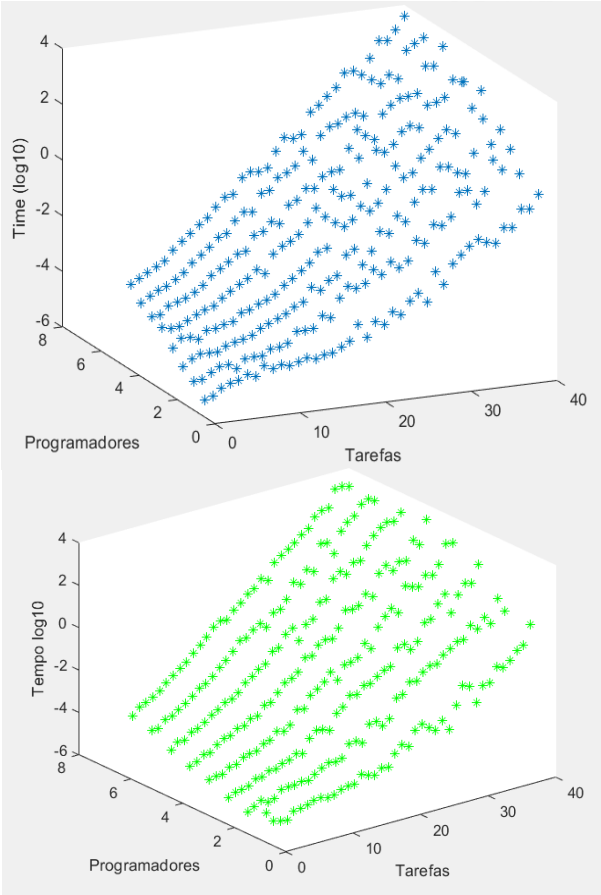


Gráfico 2 Mec.95584

Através da visualização do gráfico é fácil verificar que o tempo de execução cresce exponencialmente com o aumento do número de tarefas e de programadores. Existem regiões onde os tempos quando se aumenta em 1 tarefa ou em 1 programador não aumentam podendo por vezes até diminuirem, devendo-se isto ao método de programação adotado.

Gráfico 1 97746

No *Gráfico 2 95584* verificamos o mesmo que no *Gráfico 1 97746*, embora as tarefas tenham lucros e valores de atributos diferentes, os tempos de execução não variam muito. Sendo este exponencial à mesma, mas mais linear numa escala logarítmica. Constatamos também que com o aumento dos números existe uma maior quantidade de anomalias, com anomalias referimo-nos a tempos mais baixos ou iguais para um maior número de tarefas/programadores.

Ao olharmos para os dois gráficos vemos que as diferenças são mínimas e é evidente que o tempo de execução tem um crescimento exponencial com o aumento tanto dos programadores como das tarefas. Não é surpresa chegar a esta conclusão, porque era o esperado, devido à complexidade computacional do problema referido anteriormente.

Seguidamente apresentamos a atribuição de tarefas consoante o programador a que estas foram atribuídas e caso não tenho sido realizadas ficaram em vermelho numa linha superior. Estas figuras são em ambos os casos referentes a 40 tarefas e 8 programadores.

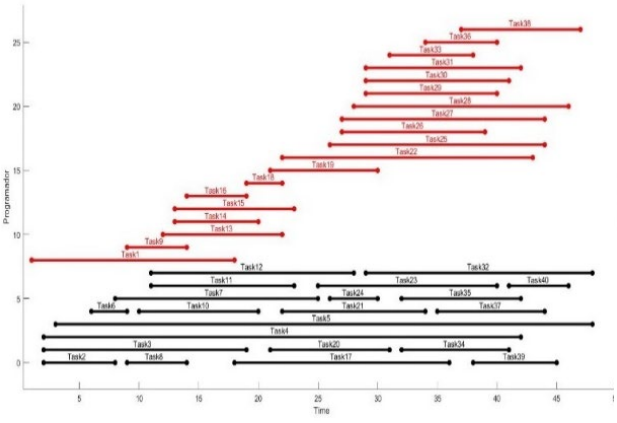


Figura 1 97746

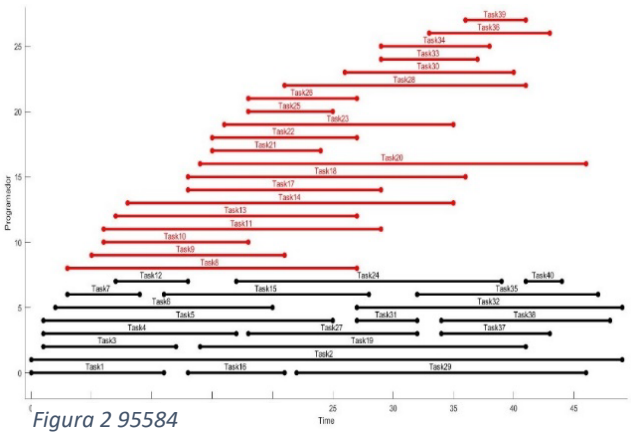


Figura 2 95584

Número máximo de tarefas

Foi-nos proposta a verificação do número máximo de tarefas possíveis a realizar sem termos em consideração o lucro proveniente destas. Para isto mudou-se apenas na compilação o valor *I* de 0 para 1. Deste modo faz-se com que os lucros não variem, como no ponto anterior, e sejam todos iguais a 1. Este valor é apenas escolhido por ser o mais simples e apresentar logo o número de tarefas realizadas, o que não aconteceria escolhendo um valor superior a 1, e se fosse inferior teríamos que alterar o código.

Fizemos um gráfico 3D para observar a variação do número máximo de tarefas concluídas à medida que o número de programadores e tarefas aumentam. E concluímos que se o número de tarefas aumenta, o número máximo de tarefas concluídas também irá aumentar e se o número de programadores aumenta o mesmo se verifica, o que já seria de esperar daí não termos se quer achado pertinente em qualquer um dos casos fazer a análise de um número de tarefas inferior ao número de programadores, uma vez que seriam sempre todas feitas e não apresenta grande relevância.

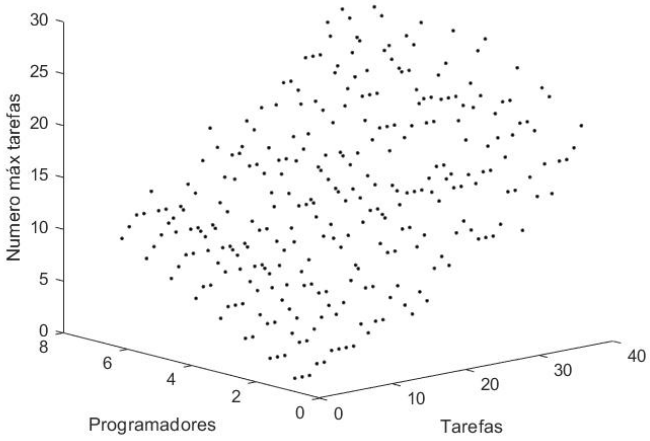


Gráfico 3 95584

Para o caso do NMEc. 95584 obtivemos um número máximo de tarefas, 27, para a situação em que são 38 tarefas e 7 programadores, o que nos surpreendeu pois esperávamos que o máximo ocorresse com o maior número de tarefas ou pelo menos com um número superior de programadores. No entanto, verificamos que para esse número de tarefas, colocando mais um programador temos menos uma tarefa a ser realizada.

Para o NMEc. 97746 também obtivemos um máximo, desta vez 28 para a mesma situação de 38 tarefas e 7 programadores.

De facto, é curioso o caso apresentado e apenas podemos tentar atribuir essa causa à forma de como as tarefas são geradas, mas tendo em atenção que estas deixaram de depender do número de programadores, esperávamos que mais programadores levasse sempre a mais tarefas realizadas.

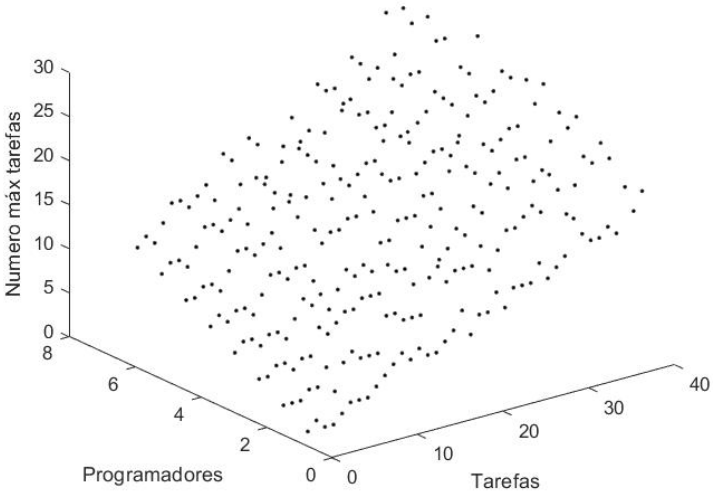
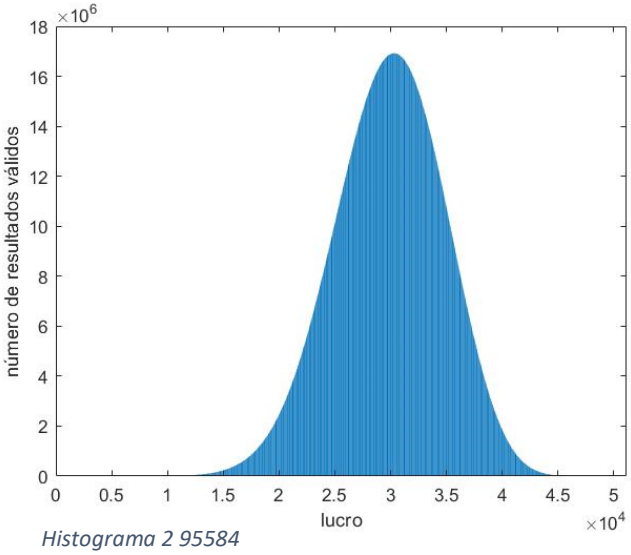
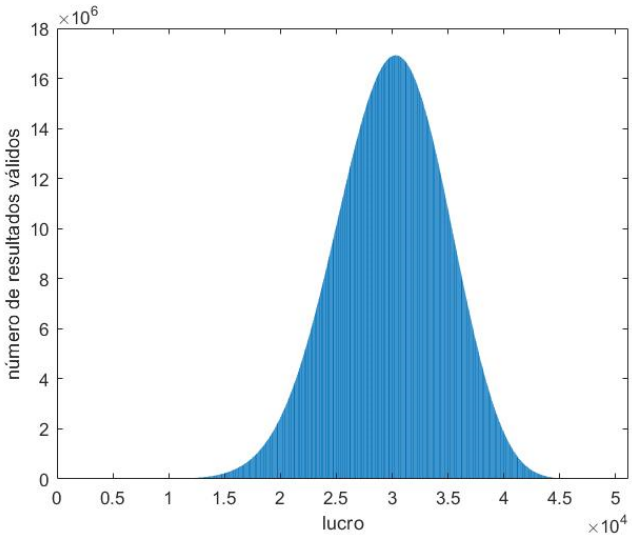


Gráfico 4 97746

Analisando os histogramas, o número de vezes que o lucro aparece na realização do problema em função do lucro, verificamos que ambos seguem uma distribuição normal.



Histograma 2 95584



Histograma 1 97746

Random_approach

O método *random_approach,* explicado anteriormente, também foi testado e comparado com os dados do método normal, *generate_possibilities*. Comparando os tempos de execução para números pequenos o método analítico é bastante mais eficiente, mas à medida que o número de programadores e tarefas aumenta este vai perdendo para a nova abordagem, assim sendo, na teoria este é o mais eficiente para números maiores. No entanto, reparámos que começa a apresentar cada vez mais erros de resultados, que embora próximos ao melhor resultado muitas das vezes fica-se por aí, apenas próximo. Este tipo de programação continua a ser válido e bom para problemas com uma resolução de uma complexidade computacional muito elevada ou mesmo impossível. Em relação à coerência do programa *random_approach*, esta poderia ser aumentada é um facto, mas com isso seria aumentado o número de testes que o programa teria de fazer, logo os tempos de execução iriam aumentar também, chegando a um ponto em que para ser quase perfeito teria de ter tempos muito superiores ao primeiro programa, nunca garantindo ter feito todas as hipóteses a não ser que fossem guardadas as combinações que foram testadas, o que seria impossível guardar em termos de memória. O número de testes poderia ir variando com a situação, mas para isso teriam que se realizar mais testes para verificar quais os valores ótimos a realizar consoante o número de programadores e tarefas.

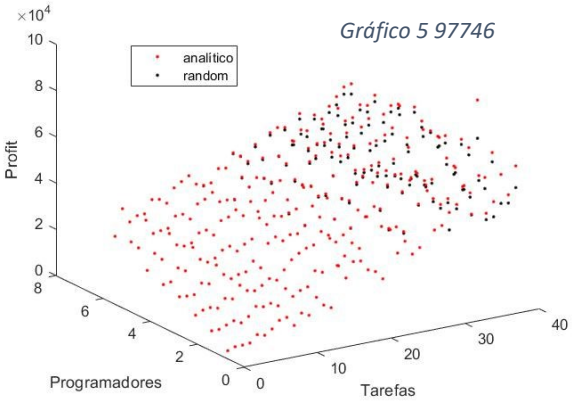


Gráfico 5 97746

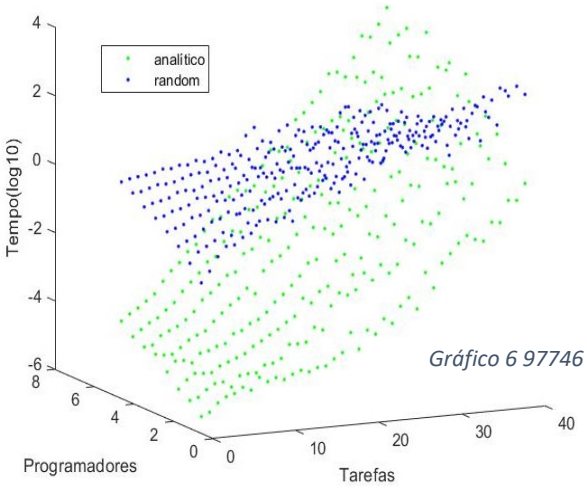


Gráfico 6 97746

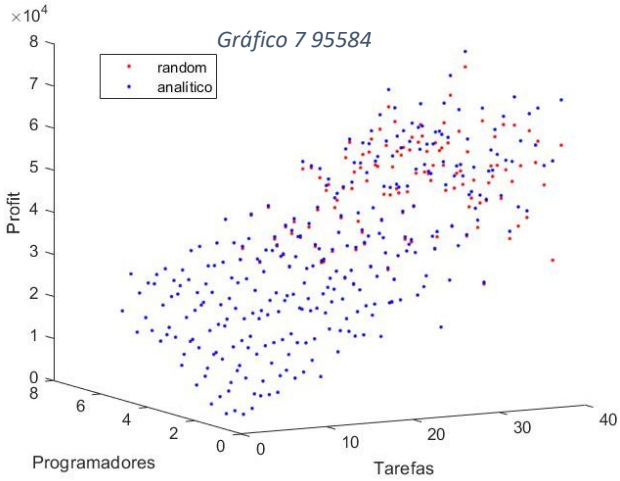


Gráfico 7 95584

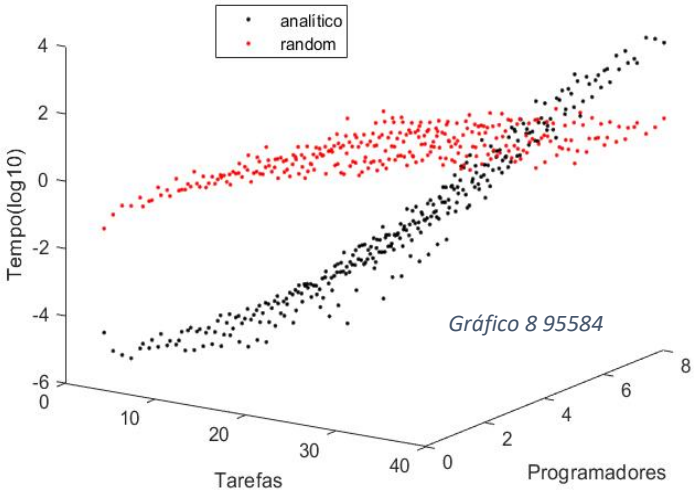


Gráfico 8 95584

Gen2

Após se ter verificado que para um programador houve grandes benefícios, como se observa no *Gráfico 9*, podemos verificar pelo gráfico que o tempo de execução desta implementação é superior até às 17 tarefas. Acreditamos que este método perca para um número inferior devido à necessidade de reorganizar as tarefas pela data de fim.

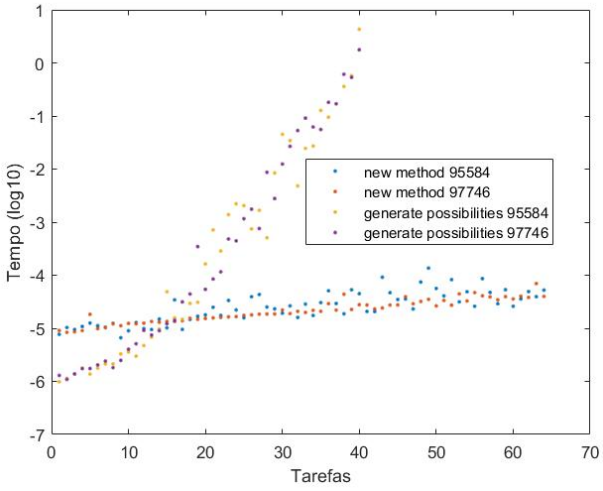


Gráfico 9

Analisando *gen2* verificámos que esta não dá o maior número máximo de tarefas a realizar, mas também se concluiu que não dá um número superior pelo que, apesar de não servir para obter uma solução perfeita conseguimos obter uma que se aproxima o suficiente. Em situações que é preferível obter uma solução razoável em um tempo muito, mas muito reduzido é uma boa decisão a tomar, sendo obviamente ainda preferível ao método *random_approach*.

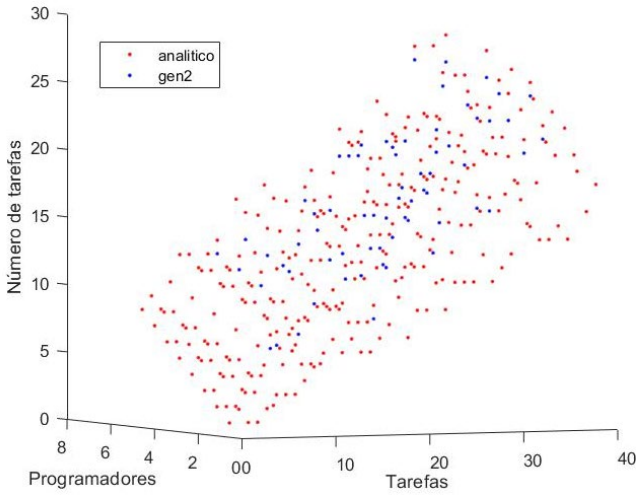


Gráfico 10 97746

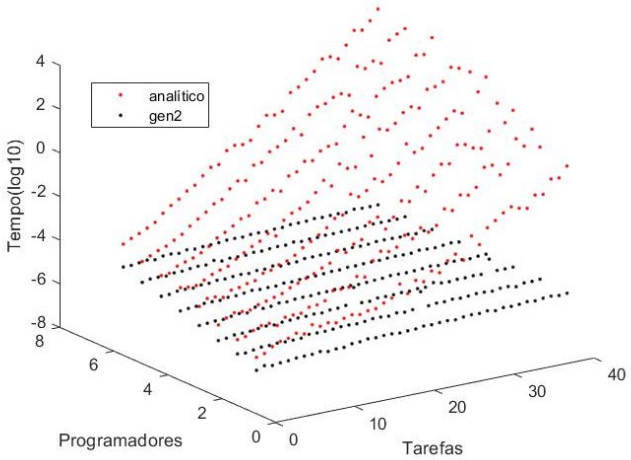


Gráfico 11 97746

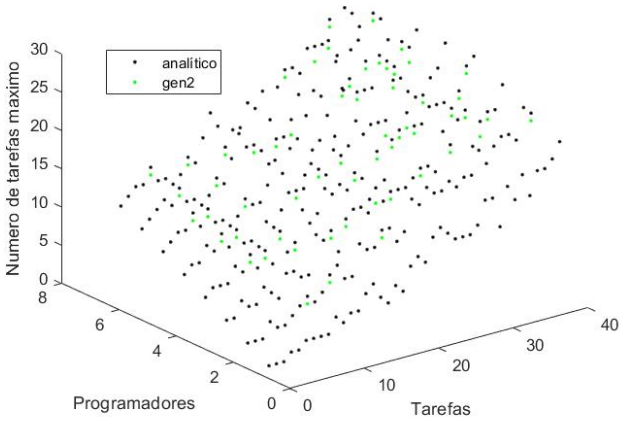


Gráfico 12 95584

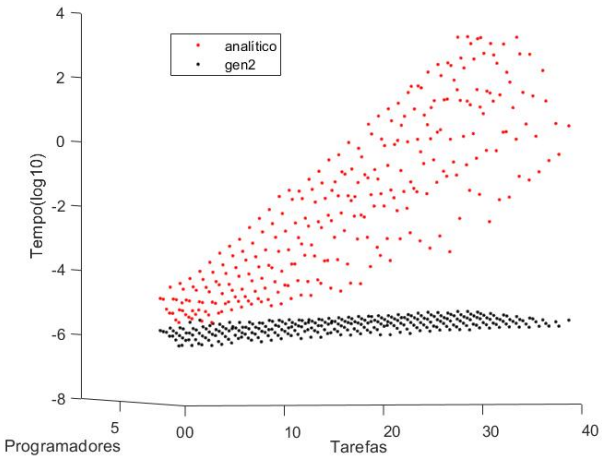


Gráfico 13 95584

Dumb_approach

Apesar de termos considerado esta ideia como inferior decidimos à mesma verificar como se iriam comparar os tempos de execução. E como podemos ver o crescimento ocorre da mesma forma. No entanto, como esperado é pior para números mais elevados, como visto anteriormente, devido ao seguimento de combinações impossíveis logo descartadas mais ao inicio pela *generate_possibilities* sendo por isso, *dumb_approach,* mais linear, realiza sempre 2^n. O mais difícil é encontrar uma explicação para *generate_possibilities* ter um desempenho pior para as situações mais pequenas, sendo que uma causa poderá ser a eventual diminuição da frequência do *boost* do processador, uma vez que este era variável. Ou ainda por causa de na função *generate_possibilities* se fazer um ciclo cada vez que o lucro é superior para a atribuição do *best_assigned_to* algo que não foi tido em consideração no desenvolvimento da função *dumb_approach*.

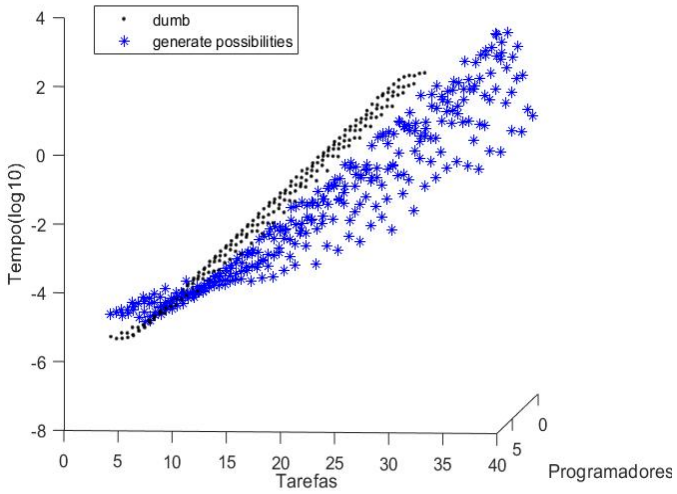


Gráfico 14 97746

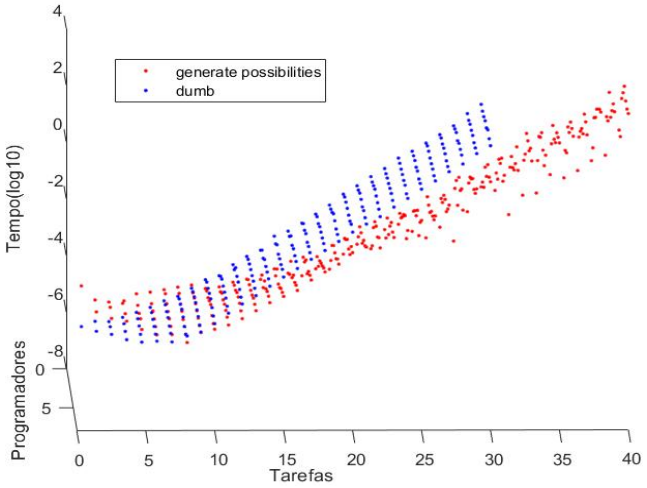


Gráfico 15 95584

Melhoramento do programa

Foi nos ainda proposto que tentássemos melhorar o programa. E como tal para o nosso código principal, *generate_possibilities*, acabámos por usar programação dinâmica, já que, o programa tem a capacidade de calcular e guardar os valores e quando decidimos não utilizar os valores calculados a seguir voltamos aos valores anteriores não tendo de voltar a calcular tudo desde o início. Julgamos que a abordagem *Divide and Conquer* seria mais benéfica na situação em que é possível fazer a computação em paralelo havendo ainda a desvantagem de alguns *cores* terem que ficar à espera que outros acabassem pois iria permitir a não seleção de opções que não são compatíveis. Sendo que nesta situação não é a melhor abordagem pois tal realização das tarefas não é a que nos ajuda a encontrar o melhor lucro possível.

Para casos mais complexos que aqui abordados também se poderia tomar outra iniciativa que seria verificar o caso do lucro máximo, isto pois comparar lucros é mais fácil que verificar se o conjunto é possível. No entanto, na prática este método não deve ser o mais eficaz pois iria depender muito do lucro das tarefas não sendo então um método garantido e que poderia apenas aumentar a complexidade computacional do problema.

Na questão do *random_approach* fazer-se um número de testes variáveis com o número de tarefas e programadores parece uma melhor ideia, pois nos casos iniciais verificamos que se chegou à resposta correta e nos finais o mesmo não se verifica, sendo possível terem ocorrido repetições de casos que leva a computação desnecessária e a um aumento de tempo como se verificou. No entanto, utilizando $2^{\text{número de tarefas}}$ é uma boa opção que quase irá garantir a melhor solução, sendo que já será preciso ter atenção ao tipo de dados utilizados e muito tempo.

Apesar de não termos implementado nenhum método mais exótico de programação, após pesquisa e diálogo em tempo letivo para a situação de um só programador em que apenas queremos analisar o número máximo de tarefas que se pode realizar, utilizou-se o método que se descreveu anteriormente que apresenta grandes benefícios como se pode constatar.

Conclusão

Com o decorrer do trabalho proposto verificámos que este poderia ter várias aplicações praticas. Observámos que seria excelente para uma empresa conseguir gerir melhor os trabalhos que tem e os programadores que vão utilizar. Por exemplo existem situações em que mais programadores não significa maio eficácia. Podemos ainda melhorar um pouco o programa para que os programadores tenham certos rendimentos por tarefas pondo os que recebem mais a fazer o menor número de tarefas e os que recebem menos o maior número. Conseguindo assim o maior lucro possível para a empresa tendo todas as variáveis consideradas.

Este programa pode ser adaptado também para outro tipo de problemas tais como na realização de turmas para uma escola, assim em vez de programadores teríamos turmas e em vez de tarefas teríamos alunos. Cada aluno teria determinadas caraterísticas e iriamos ver qual a turma mais adequada para este. Na atribuição de salas a turmas, também podemos observar uma utilidade para este algoritmo sendo que até se poderia utilizar a função *gen2* de modo a se ter uma solução muito mais rápida em que as tarefas são as aulas, hora de início e de fim, não esquecendo a capacidade da sala para a dimensão da turma.

Ainda outra implementação seria por exemplo num cinema. Em que interessa a seleção do filme com maior lucro consoante a hora a que é exibido, podendo esse ser um parâmetro variável, assim, tendo um tempo de início e de fim como as tarefas tinham, não sendo possível haver sobreposição na mesma sala.

Bibliografia

https://stackoverflow.com/questions/23449681/scheduling-the-most-intervals-rather-than-scheduling-the-most-tasks

https://cse.buffalo.edu/~hartloff/CSE331-Summer2015/greedy.pdf

https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pearson/04GreedyAlgorithms-2x2.pdf

http://www.mistaconference.org/2007/papers/Interval%20Scheduling.pdf

http://www.mistaconference.org/2007/papers/Interval%20Scheduling.pdf

https://en.wikipedia.org/wiki/Interval_scheduling#:~:text=Interval%20scheduling%20is%20a%20class,the%20area%20of%20algorithm%20design.&text=The%20interval%20scheduling%20maximization%20problem,as%20many%20tasks%20as%20possible

Código

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
// AED, 2020/2021
//
// TODO: Tiago Santos 95584
// TODO: Vasco Costa 97746
// TODO: place the student number and name here (if applicable)
//
// Brute-force solution of the generalized weighted job selection problem
//
// Compile with "cc -Wall -O2 job_selection.c -lm" or equivalent
//
// In the generalized weighted job selection problem we will solve here we have T programming tasks and P programmers.
// Each programming task has a starting date (an integer), an ending date (another integer), and a profit (yet another
// integer). Each programming task can be either left undone or it can be done by a single programmer. At any given
// date each programmer can be either idle or it can be working on a programming task. The goal is to select the
// programming tasks that generate the largest profit.
//
// Things to do:
//   0. (mandatory)
//      Place the student numbers and names at the top of this file.
//   1. (highly recommended)
//      Read and understand this code.
//   2. (mandatory)
```

```
//     Solve the problem for each student number of the group and for
//       N=1, 2, ..., as higher as you can get and
//       P=1, 2, ... min(8,N)
//     Present the best profits in a table (one table per student number).
//     Present all execution times in a graph (use a different color for the times of each student number).
//     Draw the solutions for the highest N you were able to do.
//  3. (optional)
//     Ignore the profits (or, what is the same, make all profits equal); what is the largest number of programming
//     tasks that can be done?
//  4. (optional)
//     Count the number of valid task assignments. Calculate and display an histogram of the number of occurrences of
//     each total profit. Does it follow approximately a normal distribution?
//  5. (optional)
//     Try to improve the execution time of the program (use the branch-and-bound technique).
//     Can you use divide and conquer to solve this problem?
//     Can you use dynamic programming to solve this problem?
//  6. (optional)
//     For each problem size, and each student number of the group, generate one million (or more!) valid random
//     assignments and compute the best solution found in this way. Compare these solutions with the ones found in
//     item 2.
//  7. (optional)
//     Surprise us, by doing something more!
//  8. (mandatory)
//     Write a report explaining what you did. Do not forget to put all your code in an appendix.
//
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include "elapsed_time.h"
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
// Random number generator interface (do not change anything in this code section)
//
// In order to ensure reproducible results on Windows and GNU/Linux, we use a good random number generator, available at
//   https://www-cs-faculty.stanford.edu/~knuth/programs/rng.c
// This file has to be used without any modifications, so we take care of the main function that is there by applying
// some C preprocessor tricks
//
#define main rng_main // main gets replaced by rng_main
#ifdef __GNUC__
```

```
int rng_main() __attribute__((__unused__)); // gcc will not complain if rnd_main() is not used
#endif#include "rng.c"
#undef main // main becomes main again
#define srandom(seed) ran_start((long) seed) // start the pseudo-random number generator
#define random() ran_arr_next() // get the next pseudo-random number (0 to 2^30-1)
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
// problem data (if necessary, add new data fields in the structures; do not change anything else in this code section)
//
// on the data structures declared below, a comment starting with
// * a I means that the corresponding field is initialized by init_problem()
// * a S means that the corresponding field should be used when trying all possible cases
// * IS means both (part initialized, part used)
//
#if 1
#define MAX_T 64 // maximum number of programming tasks
#define MAX_P 10 // maximum number of programmers
typedef struct {
  int starting_date; // I starting date of this task
  int ending_date; // I ending date of this task
  int profit; // I the profit if this task is performed
  int assigned_to; // S current programmer number this task is assigned to (use -1 for no assignment)
  int best_assigned_to;
}
task_t;
typedef struct {
  int NMec; // I  student number
  int T; // I  number of tasks
  int P; // I  number of programmers
  int I; // I  if 1, ignore profits
  int total_profit; // S  current total profit
  double cpu_time; // S  time it took to find the solution
  task_t task[MAX_T]; // IS task data
  int busy[MAX_P]; // S  for each programmer, record until when she/he is busy (-1 means idle)
  char dir_name[16]; // I  directory name where the solution file will be created
  char file_name[64]; // I  file name where the solution data will be stored

  int best_total_profit;
  long long valid_tasks;
  int sum_all_tasks;
  int * valid_tasks_profits;
}
```

```
problem_t;

int compare_tasks(const void * t1,
  const void * t2) {
  int d1, d2;
  d1 = ((task_t * ) t1) -> starting_date;
  d2 = ((task_t * ) t2) -> starting_date;
  if (d1 != d2)
    return (d1 < d2) ? -1 : +1;
  d1 = ((task_t * ) t1) -> ending_date;
  d2 = ((task_t * ) t2) -> ending_date;
  if (d1 != d2)
    return (d1 < d2) ? -1 : +1;
  return 0;
}

void init_problem(int NMec, int T, int P, int ignore_profit, problem_t * problem) {
  int i, r, scale, span, total_span;
  int * weight;
  //
  // input validation
  //
  if (NMec < 1 || NMec > 999999) {
    fprintf(stderr, "Bad NMec (1 <= NMex (%d) <= 999999)\n", NMec);
    exit(1);
  }
  if (T < 1 || T > MAX_T) //numero de tarefas
  {
    fprintf(stderr, "Bad T (1 <= T (%d) <= %d)\n", T, MAX_T);
    exit(1);
  }
  if (P < 1 || P > MAX_P) //numero de programadores
  {
    fprintf(stderr, "Bad P (1 <= P (%d) <= %d)\n", P, MAX_P);
    exit(1);
  }
  //
  // the starting and ending dates of each task satisfy 0 <= starting_date <= ending_date <= total_span
  //
  total_span = (10 * T + P - 1) / P;
  if (total_span < 30)
    total_span = 30;
```

```
//
// probability of each possible task duration
//
// task span relative probabilities
//
// |  0  0  4  6  8 10 12 14 16 18 | 20 | 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1 | smaller than 1
// |  0  0  2  3  4  5  6  7  8  9 | 10 | 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 | 30 31 ... span
//
weight = (int * ) alloca((size_t)(total_span + 1) * sizeof(int)); // allocate memory (freed automatically)
if (weight == NULL) {
 fprintf(stderr, "Strange! Unable to allocate memory\n");
 exit(1);
}
#define sum1(298.0) // sum of weight[i] for i=2,...,29 using the data given in the comment above
#define sum2((double)(total_span - 29)) // sum of weight[i] for i=30,...,data_span using a weight of 1
#define tail 100
scale = (int) ceil((double) tail * 10.0 * sum2 / sum1); // we want that scale*sum1 >= 10*tail*sum2, so that large task
if (scale < tail) // durations occur 10% of the time
 scale = tail;
weight[0] = 0;
weight[1] = 0;
for (i = 2; i <= 10; i++)
 weight[i] = scale * (2 * i);
for (i = 11; i <= 29; i++)
 weight[i] = scale * (30 - i);
for (i = 30; i <= total_span; i++)
 weight[i] = tail;
#undef sum1
#undef sum2
#undef tail
//
// accumulate the weigths (cummulative distribution)
//
for (i = 1; i <= total_span; i++)
 weight[i] += weight[i - 1];
//
// generate the random tasks
//
//srandom(NMec + 314161 * T + 271829 * P);
srandom(NMec + 314161 * T); //para ser interessante a comparação sem o numero de programadores afetar
problem -> NMec = NMec;
problem -> T = T;
```

```
problem -> P = P;
problem -> I = (ignore_profit == 0) ? 0 : 1;
for (i = 0; i < T; i++) {
 //
 // task starting an ending dates
 //
 r = 1 + (int) random() % weight[total_span]; // 1 .. weight[total_span]
 for (span = 0; span < total_span; span++)
  if (r <= weight[span])
    break;
 problem -> task[i].starting_date = (int) random() % (total_span - span + 1);
 problem -> task[i].ending_date = problem -> task[i].starting_date + span - 1;
 //
 // task profit
 //
 // the task profit is given by r*task_span, where r is a random variable in the range 50..300 with a probability
 //   density function with shape (two triangles, the area of the second is 4 times the area of the first)
 //
 //     *
 //    /|   *
 //   / |      *
 //  /  |         *
 // *---*----------*
 // 50 100 150 200 250 300
 //
 scale = (int) random() % 12501; // almost uniformly distributed in 0..12500
 if (scale <= 2500)
   problem -> task[i].profit = 1 + round((double) span * (50.0 + sqrt((double) scale)));
 else
   problem -> task[i].profit = 1 + round((double) span * (300.0 - 2.0 * sqrt((double)(12500 - scale))));
}
//
// sort the tasks by the starting date
//
qsort((void * ) & problem -> task[0], (size_t) problem -> T, sizeof(problem -> task[0]), compare_tasks);
//
// finish
//
if (problem -> I != 0)
  for (i = 0; i < problem -> T; i++)
    problem -> task[i].profit = 1;
#define DIR_NAME problem -> dir_name
```

```
  if (snprintf(DIR_NAME, sizeof(DIR_NAME), "%06d", NMec) >= (int) sizeof(DIR_NAME)) {
    fprintf(stderr, "Directory name too large!\n");
    exit(1);
  }
  #undef DIR_NAME
  #define FILE_NAME problem -> file_name
  if (snprintf(FILE_NAME, sizeof(FILE_NAME), "%06d/%02d_%02d_%d.txt", NMec, T, P, problem -> I) >= (int) sizeof(FILE_NAME)) {
    fprintf(stderr, "File name too large!\n");
    exit(1);
  }
  #undef FILE_NAME
}
#endif
////////////////////////////////////////////////////////////////////////////////
void dumb_approach(problem_t * problem, int t[], int tarefa_atual) {
  if (problem -> T == tarefa_atual) {
    for (int i = 0; i < tarefa_atual; i++) {
      if (t[i] == 1) {
        for (int j = 0; j < problem -> P; j++) {
          if (problem -> busy[j] < problem -> task[i].starting_date) {
            problem -> task[i].assigned_to = j;
            problem -> busy[j] = problem -> task[i].ending_date;
            problem -> total_profit += problem -> task[i].profit;
            break;
          }
        }
        if (problem -> task[i].assigned_to == -1) {
          problem -> total_profit = 0;
          for (int j = 0; j < problem -> P; j++)
            problem -> busy[j] = -1;
          for (int j = 0; j < problem -> T; j++)
            problem -> task[j].assigned_to = -1;
          return;
        }
      }
    }
    problem -> valid_tasks++;
    if (problem -> total_profit > problem -> best_total_profit) {
      problem -> best_total_profit = problem -> total_profit;
      for (int i = 0; i < problem -> T; i++)
        problem -> task[i].best_assigned_to = problem -> task[i].assigned_to;
    }
```

```
    problem -> total_profit = 0;
    for (int j = 0; j < problem -> P; j++)
      problem -> busy[j] = -1;
    for (int j = 0; j < problem -> T; j++)
      problem -> task[j].assigned_to = -1;
    return;
  } else {
    t[tarefa_atual - 1] = 0; //nao considerar
    dumb_approach(problem, t, tarefa_atual + 1);
    t[tarefa_atual - 1] = 1; //considerar
    dumb_approach(problem, t, tarefa_atual + 1);
  }
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
// problem solution (place your solution here)
//
int compare_tasks_2(const void * t1,
  const void * t2) {
  int d1, d2;
  d1 = ((task_t * ) t1) -> ending_date;
  d2 = ((task_t * ) t2) -> ending_date;
  if (d1 != d2)
    return (d1 < d2) ? -1 : +1;
  d1 = ((task_t * ) t1) -> starting_date;
  d2 = ((task_t * ) t2) -> starting_date;
  if (d1 != d2)
    return (d1 < d2) ? -1 : +1;
  return 0;
}
void gen2(problem_t * problem) {
  problem -> best_total_profit = 0;
  problem -> valid_tasks = 0;
  problem -> total_profit = 0;
  for (int i = 0; i < problem -> T; i++) //meter as tarefas sem serem atribuidas
  {
    problem -> task[i].assigned_to = -1;
  }
  for (int i = 0; i < problem -> P; i++) //estão todos sem data de começo
  {
    problem -> busy[i] = -1;
  }
```

```
 qsort((void * ) & problem -> task[0], (size_t) problem -> T, sizeof(problem -> task[0]), compare_tasks_2);
 for (int prog = 0; prog < problem -> P; prog++)
  for (int i = 0; i < problem -> T; i++) {
   if (problem -> task[i].assigned_to == -1) {
    if (problem -> busy[prog] < problem -> task[i].starting_date) {
     problem -> best_total_profit++;
     problem -> task[i].assigned_to = prog;
     problem -> task[i].best_assigned_to = prog;
     problem -> busy[prog] = problem -> task[i].ending_date;
    }
   }
  }
}
void generate_possibilities(problem_t * problem, int tarefa_atual) //tarefa_atual inicial 0
{
 if (tarefa_atual < problem -> T) {
  if (tarefa_atual == 0) {
   problem -> best_total_profit = 0;
   problem -> valid_tasks = 0;
   problem -> total_profit = 0;

   for (int i = 0; i < problem -> T; i++) //meter as tarefas sem serem atribuidas
   {
    problem -> task[i].assigned_to = -1;
    problem -> sum_all_tasks += problem -> task[i].profit;
   }
   problem -> valid_tasks_profits = (int * ) calloc(problem -> sum_all_tasks + 1, sizeof(int));
   for (int i = 0; i < problem -> P; i++) {
    problem -> busy[i] = -1;
   }
  }

  //new method
  #if 1
  if (problem -> I == 1 && problem -> P == 1) //se o lucro não importar e for um programador, aquilo especial
  {
   qsort((void * ) & problem -> task[0], (size_t) problem -> T, sizeof(problem -> task[0]), compare_tasks_2);
   for (int i = 0; i < problem -> T; i++) {
    if (problem -> busy[0] < problem -> task[i].starting_date) {
     problem -> best_total_profit++;
     problem -> task[i].assigned_to = 0;
     problem -> task[i].best_assigned_to = 0;
```

```
      problem -> busy[0] = problem -> task[i].ending_date;
    }
   }
   return;
  }
  #endif

  // se nao for para contar esta tarefa fazemos logo a proxima tarefa.
  generate_possibilities(problem, tarefa_atual + 1);
  //se for para contar esta tarefa fazemos todos os incrementeos necessarios e chamamos a proxima tarefa.
  int busy_copy;
  int total_profit_copy = problem -> total_profit;
  int assigned_to_copy = problem -> task[tarefa_atual].assigned_to;
  int i;

  for (i = 0; i < problem -> P; i++) {
   if (problem -> busy[i] < problem -> task[tarefa_atual].starting_date) {
    busy_copy = problem -> busy[i];
    problem -> busy[i] = problem -> task[tarefa_atual].ending_date;
    problem -> task[tarefa_atual].assigned_to = i;
    problem -> total_profit += problem -> task[tarefa_atual].profit;
    break;
   }
  }
  if (problem -> task[tarefa_atual].assigned_to == -1) //corta se o ramo
  {
   problem -> busy[i] = busy_copy;
   problem -> total_profit = total_profit_copy;
   return;
  }
  generate_possibilities(problem, tarefa_atual + 1);

  problem -> task[tarefa_atual].assigned_to = assigned_to_copy;
  problem -> busy[i] = busy_copy;
  problem -> total_profit = total_profit_copy;
 } else if (tarefa_atual == problem -> T) {
  problem -> valid_tasks++;
  problem -> valid_tasks_profits[problem -> total_profit]++;
  if (problem -> total_profit > problem -> best_total_profit) {
   problem -> best_total_profit = problem -> total_profit;
   for (int i = 0; i < problem -> T; i++)
    problem -> task[i].best_assigned_to = problem -> task[i].assigned_to;
```

```
    }
  }
}

void random_approach(problem_t * problem) {
  problem -> best_total_profit = 0;
  problem -> valid_tasks = 0;
  while (problem -> valid_tasks < 1200000) {
    problem -> total_profit = 0;

    for (int i = 0; i < problem -> P; i++) //estão todos sem data de começo
    {
      problem -> busy[i] = -1;
    }

    // vamos percorrendo as tarefas e faz se binariamente se seleciona ou se nao a tarefa
    int bin;
    for (int i = 0; i < problem -> T; i++) {
      problem -> task[i].assigned_to = -1;
      bin = rand() % 2; //dá 0 ou 1
      if (bin) {
        int p;
        for (p = 0; p < problem -> P; p++) {
          if (problem -> busy[p] < problem -> task[i].starting_date) {
            problem -> busy[p] = problem -> task[i].ending_date;
            problem -> task[i].assigned_to = p;
            problem -> total_profit += problem -> task[i].profit;
            break;
          }
        }
        if (problem -> task[i].assigned_to == -1) {
          problem -> total_profit = -1;
          problem -> valid_tasks--;
          break;
        }
      }
    }
    problem -> valid_tasks++;
    if (problem -> best_total_profit < problem -> total_profit) {
      problem -> best_total_profit = problem -> total_profit;
      for (int i = 0; i < problem -> T; i++)
        problem -> task[i].best_assigned_to = problem -> task[i].assigned_to;
```

```
    }
  }
}
#if 1

static void solve(problem_t * problem) {
  FILE * fp;
  int i;
  //
  // open log file
  //
  //(void)mkdir(problem->dir_name,S_IRUSR | S_IWUSR | S_IXUSR);
  (void) mkdir(problem -> dir_name); //  o de cima nao me compila em windows
  fp = fopen(problem -> file_name, "w");
  if (fp == NULL) {
    fprintf(stderr, "Unable to create file %s (maybe it already exists? If so, delete it!)\n", problem -> file_name);
    exit(1);
  }
  //
  // solve
  //
  problem -> cpu_time = cpu_time();
  // call your (recursive?) function to solve the problem here
  #if 0
  int arr[problem -> T];
  problem -> best_total_profit = 0;
  problem -> valid_tasks = 0;
  problem -> total_profit = 0;
  for (int i = 0; i < problem -> P; i++)
    problem -> busy[i] = -1;
  for (int i = 0; i < problem -> T; i++)
    problem -> task[i].assigned_to = -1;
  dumb_approach(problem, arr, 1);
  #endif

  #if 0
  generate_possibilities(problem, 0);
  #endif

  #if 0
  //apenas para ver o numero maximo de tarefas
  gen2(problem);
```

```
 #endif

 #if 1
 random_approach(problem);
 #endif

 problem -> cpu_time = cpu_time() - problem -> cpu_time;
 //
 // save solution data
 //
 fprintf(fp, "NMec = %d\n", problem -> NMec);
 fprintf(fp, "T = %d\n", problem -> T);
 fprintf(fp, "P = %d\n", problem -> P);
 fprintf(fp, "Profits%s ignored\n", (problem -> I == 0) ? " not" : "");
 fprintf(fp, "Valid tasks: %I64d\n", problem -> valid_tasks); //isto esta certo mas se tiveres outro melhor que lld...
 fprintf(fp, "Solution time = %.3e\n", problem -> cpu_time);
 fprintf(fp, "Task data\n");
 #define TASK problem -> task[i]
 for (i = 0; i < problem -> T; i++)
  fprintf(fp, "  %3d %3d %5d %d\n", TASK.starting_date, TASK.ending_date, TASK.profit, TASK.best_assigned_to);
 #undef TASK

 fprintf(fp, "Best profit: %d\n", problem -> best_total_profit);

 #if 0
 fprintf(fp, "All profits:\n");
 for (int i = 0; i < problem -> sum_all_tasks; i++)
  if (problem -> valid_tasks_profits[i] != 0)
   fprintf(fp, "%d %d\n", i, problem -> valid_tasks_profits[i]);
 #endif

 fprintf(fp, "End\n");
 //
 // terminate
 //
 if (fflush(fp) != 0 || ferror(fp) != 0 || fclose(fp) != 0) {
  fprintf(stderr, "Error while writing data to file %s\n", problem -> file_name);
  exit(1);
 }
}

#endif
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
// main program
//

int main(int argc, char ** argv) {
  problem_t problem;
  int NMec, T, P, I;

  NMec = (argc < 2) ? 2020 : atoi(argv[1]);
  T = (argc < 3) ? 5 : atoi(argv[2]);
  P = (argc < 4) ? 2 : atoi(argv[3]);
  I = (argc < 5) ? 0 : atoi(argv[4]);
  init_problem(NMec, T, P, I, & problem);
  solve( & problem);
  return 0;}
```