Procura e Planeamento

Campus da Alameda
Projeto (2018/2019)

Número do grupo: 06

Nome: Margarida Costa **Número**: 83425

Nome: Mariana Francisca Carrilho Loureiro **Número**: 83520

Classificação:

Soma das horas gastas exclusivamente para fazer este trabalho: 20 horas

Índice

1. Introdução 3
2. Modelação do *Constraint Satisfaction Problem* 4
   2.1. Estrutura do CSP 4
   2.2. Geração de Sucessores 4
   2.3. Verificação das restrições 4
   2.4. Heurísticas 5
3. Estratégias de procura implementadas 7
   3.1. Sondagem Iterativa 7
   3.2. *Iterative Limited Depth Search* (ILDS) 8
   3.3. Procura alternativa – *Depth-Bounded Discrepancy Search* (DDS) 10
4. Comparação de resultados 13
   4.1. Melhor Abordagem 14
   4.2. Melhorias a implementar 14
5. Conclusão 15
6. Bibliografia 16

1. Introdução

O problema a que este projeto pretende dar resposta consiste em obter uma afetação completa das tarefas de condução de veículos de transporte, a realizar pelos trabalhadores durante um dia de trabalho, a turnos de serviço, respeitando todas as restrições do problema.

Para esse propósito, foi modelado um problema de satisfação de restrições, sobre o qual foram posteriormente executadas várias estratégias de procura. Este relatório analisa os resultados desses algoritmos. Os valores calculados pela execução do código do projeto, e apresentados abaixo, foram registados por um *core Intel i7* a 4.00GHz.

2. Modelação do *Constraint Satisfaction Problem*

2.1. Estrutura do CSP

Foi criada uma estrutura da linguagem *Lisp* para representar o conceito de *estado* na procura, com uma abordagem construtiva em mente. É composto pelos atributos *variáveis por atribuir* e *turnos com as tarefas já atribuídas*, que se representam em lista.

Adicionalmente, a estrutura mantém (para cada estado) os valores de *custo* (como definido no enunciado – somatório das durações temporais dos turnos definidos nesse estado, com um mínimo de 6 horas para cada turno), *número de turnos definidos*, *número de viagens sem serviço na atribuição dos turnos*, *número total de tarefas no problema* e uma variável para indicar se nesse estado foi acrescentado um *novo turno* em relação ao estado-pai, e a duração das tarefas que ainda não foram atribuídas.

A função de *custo* devolve o valor de custo descrito acima, e foram também definidas as funções *objetivo* e *igualdade de estados*, para além da função de *geração de sucessores* descrita em baixo. Desta forma, é possível testar todas as técnicas de procura fornecidas no ficheiro *procura.lisp*.

2.2. Geração de Sucessores

O CSP foi modelado de forma a permitir ser explorado como um problema de otimização. Nesta perspetiva, a geração de sucessores é feita *incluindo* a verificação de restrições, para a maioria das procuras. Para a procura com as estratégias de *ILDS* e *DDS*, são gerados todos os estados-filhos potenciais, mesmo que violem restrições.

As tarefas são ordenadas temporalmente no início da execução do projeto, pelo que a solução modelada nunca assumiu um sucessor que atribuísse uma tarefa numa posição anterior a outra tarefa previamente atribuída – a nova tarefa é sempre colocada no final do novo turno.

Cada estado gera filhos atribuindo a nova tarefa a cada um dos turnos já existentes (um novo filho por turno), mais um estado-filho adicional em que a tarefa é a primeira num novo turno gerado.

2.3. Verificação das restrições

Para todas as procuras exceto *ILDS* e *DDS*, a verificação de restrições é feita durante a fase de geração de sucessores, de forma a imediatamente descartar a geração de estados inválidos. Todos estes estados-filhos são devolvidos apenas após a validação das restrições, e todos eles respeitam todas as restrições. Para o *ILDS* e o *DDS*, a validação é feita no final de cada iteração da estratégia.

O teste das restrições é efetuado para cada turno do estado proposto, e é durante esta fase que se calculam e definem, para cada estado, o *número de viagens sem serviço na atribuição dos turnos* e o *custo*. O turno é iterado uma só vez para todo este processo. As consistências espacial e temporal são também validadas durante a iteração. Desta forma, a complexidade desta fase é linear.

Restrições:

1. Todos os turnos devem começar no local L1;
2. A duração máxima de um turno de serviço é de 8:00;

3. A duração de um turno de serviço é calculada do início do serviço até ao fim do serviço. Se a duração for menos de 6 horas, então conta como se tivesse 6 horas;
4. Os turnos de serviço devem ter no máximo uma pausa para refeição;
5. Uma pausa de refeição deve ter a duração de 40 minutos;
6. O trabalhador não pode tomar refeição enquanto está a ser transportado;
7. O tempo de condução antes de ser necessário uma pausa de refeição não pode exceder as 4:00;
8. O tempo de condução inclui potenciais espaços entre tarefas que não sejam tomadas de refeição, isto é conta-se todo o tempo desde o inicio da primeira tarefa do bloco de tarefas até ao fim da última tarefa do bloco de tarefas. Nesta contabilização, no caso de a primeira tarefa do bloco ser a primeira tarefa do turno e o seu local de início não ser L1, deve-se incluir o tempo de deslocar o trabalhador de L1 até esse local (é necessário usar o mesmo raciocínio para a última tarefa de um turno e o seu local de fim).

2.4. Heurísticas

$$H1(estado) = \text{peso} * numVar(estado)$$

$$H2(estado) = peso * tempo_{tarefas}$$

A heurística 1 é admissível. A variável *numVar* corresponde ao número de turnos por atribuir no estado. O peso desta heurística no cálculo do *f* foi variado propositadamente de forma a que a procura não fosse ineficientemente focada na largura da árvore: assim, o peso é máximo na raiz da árvore, e mínimo nas folhas. A heurística é também calculada com a mesma ordem de magnitude do custo.

A heurística 2 também é admissível. A variável tempo $tempo_{tarefas}$ que calcula o tempo que falta para chegar ao nó objetivo. Inicialmente, corresponde ao tempo de duração de todas as tarefas dadas para o problema inicial. À medida que as tarefas vão sendo atribuídas aos turnos, este valor vai diminuindo.

As heurísticas encontradas e definidas acima foram testadas para 5 problemas fornecidos para teste na página da cadeira, com a estratégia *A**. Os resultados foram registados nas seguintes tabelas:

Problema\Heurística	**Heurística 1 (H1)**	**Heurística 2 (H2)**
1	9776	9776
2	21686	21686
3	33536	33536
4	65235	65235
5	91221	91221

Tabela 1 – Custo da solução

Problema\Heurística	**Heurística 1 (H1)**	**Heurística 2 (H2)**
1	0.15625	0.15625
2	1.453125	1.5
3	6.625	6.71875
4	44.828125	46.53125
5	90.8125	92.359375

Tabela 2 – Tempo (s) de execução

Problema\Heurística	**Heurística 1 (H1)**	**Heurística 2 (H2)**
1	83	83
2	170	170
3	291	291
4	569	569
5	710	710

Tabela 3 – Nós expandidos na execução

Problema\Heurística	**Heurística 1 (H1)**	**Heurística 2 (H2)**
1	202	202
2	433	433
3	861	861
4	1774	1774
5	2822	2822

Tabela 4 – Nós gerados na execução

As duas heurísticas devolvem valores muito semelhantes, com um desempenho ligeiramente melhor atribuído à heurística *H1*.

3. Estratégias de procura implementadas

Os algoritmos foram implementados de forma a enviar várias sondas ao longo de um tempo útil para explorar o máximo de caminhos possíveis, tendo como objetivo encontrar o caminho com menor custo.

3.1. Sondagem Iterativa

```
1.  ISP(node):
2.     solucao_optima = NULL
3.     while t_util:
4.        solucao = ISPProb(node)
5.        if solucao_optima == NULL:
6.           solucao_optima = solucao
7.        else if custo(solucao_optima) > custo(solucao):
8.           solucao_optima = solucao
9.    return solucao
```

Figura 1. Algoritmo para a Sondagem Iterativa

```
1.  ISPProb(node):
2.    if isGoal(node) return node
3.    if fail(node) return nil
4.    while t_util:
5.     S = generate-successors(node)
6.     index = generate-random-number(0, S_size - 1)
7.     ISPProb(S[index])
```

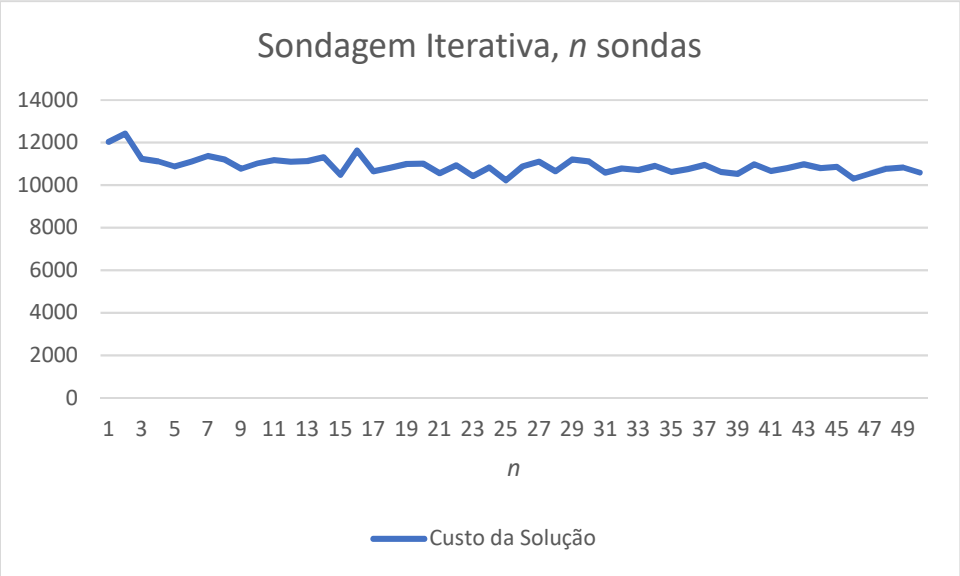
Figura 2. Função auxiliar para o algoritmo de Sondagem Iterativa

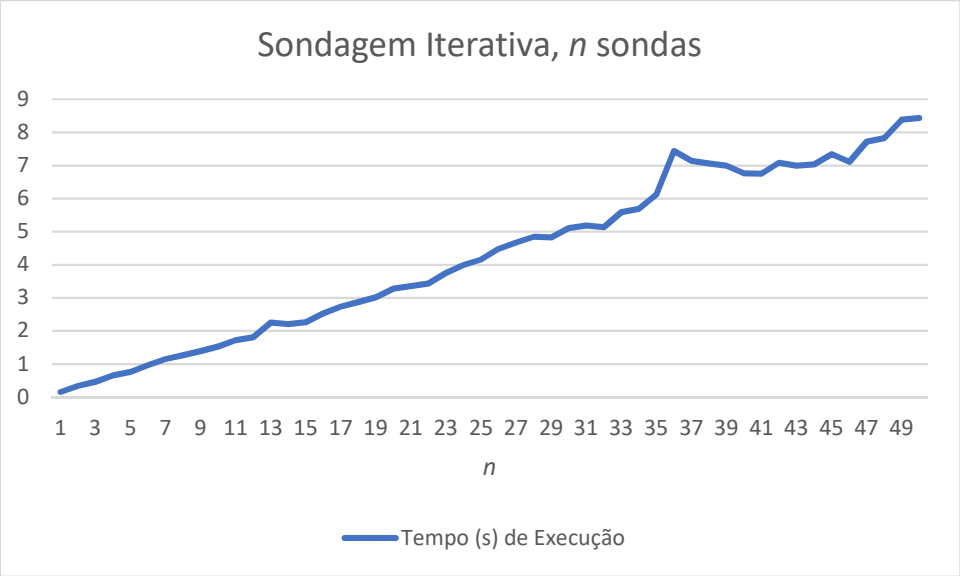
Foi implementado a versão de procura em árvore. Se o problema for pequeno suficiente, é possível explorar todo o espaço de estados, encontrando a solução óptima.

Na figura 1, o parâmetro *node* é o nó inicial do problema. A função ISP chama a função ISPNode para explorar caminhos a partir do nó raíz até ao nó objectivo. Num tempo definido, procura o máximo de caminhos que conseguir, comparando os seus custos e guardando aquele com custo menor. Ao encontrar um nó objetivo, retrocede para o nó raiz.

Na figura 2, o parâmetro *node* é o estado que está a ser de momento explorado. Na linha 1, se o estado atual é objetivo este é retornado. Na linha 2 se não se consegue gerar sucessores, retorna-se NIL. A função ***generate-successors*** expande o nó, gerando os seus sucessores. A função ***generate-random-number*** devolve um índice aleatório para aceder a S, lista de sucessores. Desta forma, escolhe-se aleatoriamente um sucessor para ser o novo estado a explorar.

Foram experimentados vários valores para o número de sondas usadas no algoritmo, de forma a definir o *n* mais eficiente para encontrar a solução ótima. Em baixo, os gráficos demonstram o comportamento obtido para diferentes valores, para um único teste:





O número de sondas não informa significativamente o desempenho do algoritmo, um resultado da aleatoriedade do algoritmo. Há, no entanto, uma tendência ligeira negativa notável no gráfico. O valor que melhor equilibra eficiência no tempo de execução da estratégia com a obtenção da solução ótima estará entre n=7 e n=11, pelo que o valor selecionado foi n=10.

3.2. *Iterative Limited Depth Search* (ILDS)

```
1.  ILDS(node, n):
2.    solucao_optima = NULL
3.    for k = 0 to n:
4.        solucao = ILDSProb(node, k, rDepth)
5.        if solucao_optima == NULL:
6.           solucao_optima = solucao
7.        else if custo(solucao_optima) > custo(solucao):
8.           solucao_optima = solucao
9.    return solucao_optima
```

Figura 3. Algoritmo para a Improved-Limited Discrepancy Search

```
1  ILDSProb(node, k, rDepth):
2  if isGoal(node) return node
3  if fail(node) return nil
4  S = generate-successors(node)
5  best_suc = choose-best-successor(S)
6  worst_suc = disjunction(best_suc)
7  if rDepth > k:
8        ILDSProb(best_suc, k, rDepth - 1)
9     if rDepth <= k:
          ILDSProb (first(worst_suc) ,k-1,rDepth-1)
```

Figura 4. Algoritmo para a Improved-Limited Discrepancy Search

A iteração para k discrepâncias gera todos os caminhos com k ou menos ramos (linha 2, figura 3). Para cada iteração, compara-se qual o caminho encontrado com menor custo.

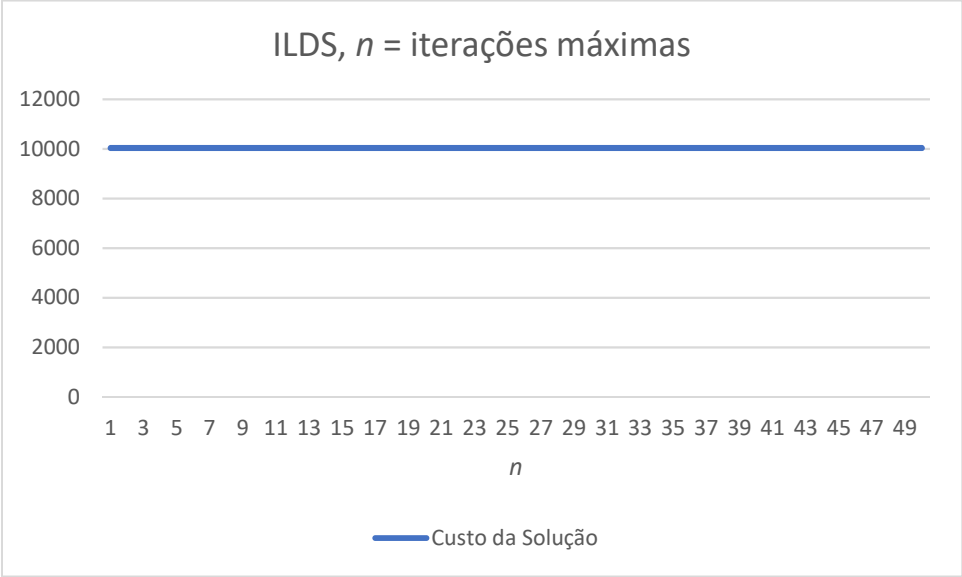
O parâmetro n do algoritmo ILDS, é o número de discrepâncias que se pode fazer. A função LDS chama a função ILDSProbe, aumentando o número de discrepâncias k até uma solução ser encontrada ou o máximo de discrepâncias ser feita. O máximo de discrepâncias corresponde à profundidade máxima do problema.

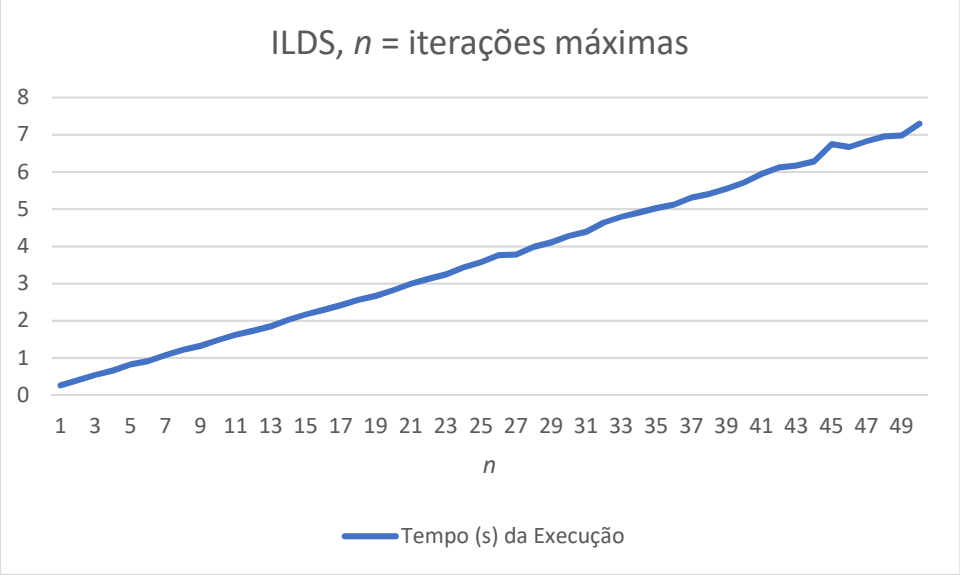
O parâmetro *rDepth* corresponde à profundidade ainda por explorar abaixo do nó atual, onde as discrepâncias que ainda podem ser tomadas.

Se a profundidade ainda por explorar for maior que k, segue-se a heurística. Atualiza-se a profundidade ainda por explorar para menos 1 unidade. Adia-se, portanto, a decisão de tomar a discrepância. Se k for maior que 0, não se segue a heurística. Diminui-se o k e a profundidade ainda por explorar em 1 unidade. Estas verificações impedem que o mesmo nó seja re-expandido entre iterações.

Para transformar o problema numa árvore binária, escolhe-se o primeiro sucessor da lista de sucessores que não foram escolhidos pela heurística.

Foram experimentados vários valores para o número de iterações máximas usadas no algoritmo, de forma a definir o *n* mais eficiente para encontrar a solução ótima. Em baixo, os gráficos demonstram o comportamento obtido para diferentes valores, para um único teste e heurística *H1*:





O número de sondas não informa o desempenho do algoritmo, pelo que o valor selecionado foi n=1, já que uma única iteração devolve a melhor solução que a estratégia acaba por encontrar. Como este é um problema de otimização, o primeiro nó com que o ILDS se depara é válido; como ele se guia por uma heurística, é também teoricamente ótimo. Por essa razão, a exploração de discrepâncias não é frutífera.

3.3. Procura alternativa – *Depth-Bounded Discrepancy Search* (DDS)

```
1.  DDS(node, n):
2.    solucao_optima = NULL
3.    for k = 0 to n:
4.        solucao = DDSProb(node, k,0)
5.        if solucao_optima == NULL:
6.           solucao_optima = solucao
          else if custo(solucao_optima) > custo(solucao):
             solucao_optima = solucao
      return solucao_optima
```

Figura 4. Algoritmo DDS

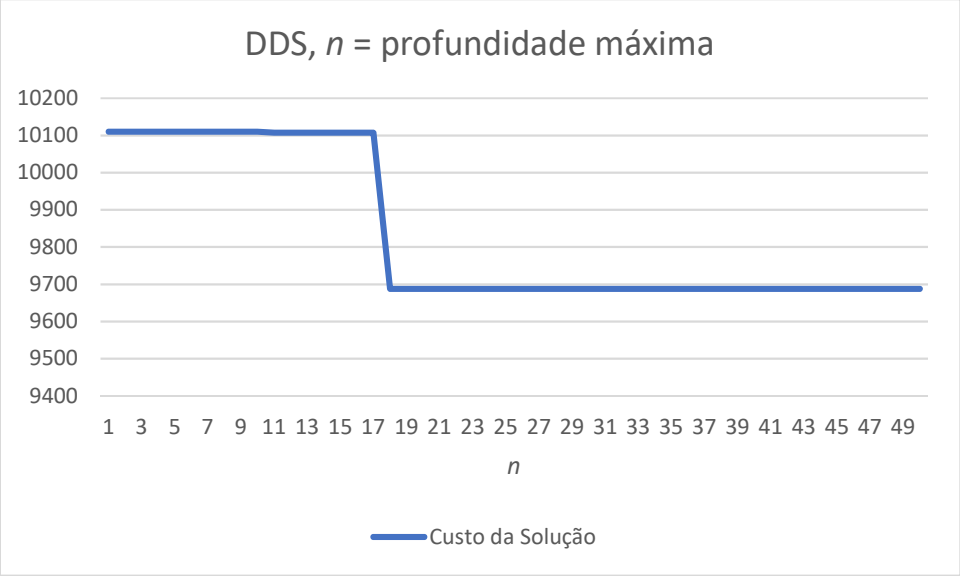
```
1.  DDSProb(node, k, prof):
2.     if isGoal(node) return node
3.     if fail(node) return nil
4.     S = generate-successors(node)
5.     best_suc = choose-best-successor(S)
6.     worst_S = disjunction(best_suc)
7.     if  k = 0:
8.        DDSProb(best_suc, k)
9.     if k = 1:
10.        DDSProb (worst_suc,k-1)
11.    if k > 1:
12.        result = DDSProb(best_suc, k-1, prof+1)
13.        if isGoal(result) return result
14.        else
15.          DDSProb(worst_S, k-1, prof + 1)
```

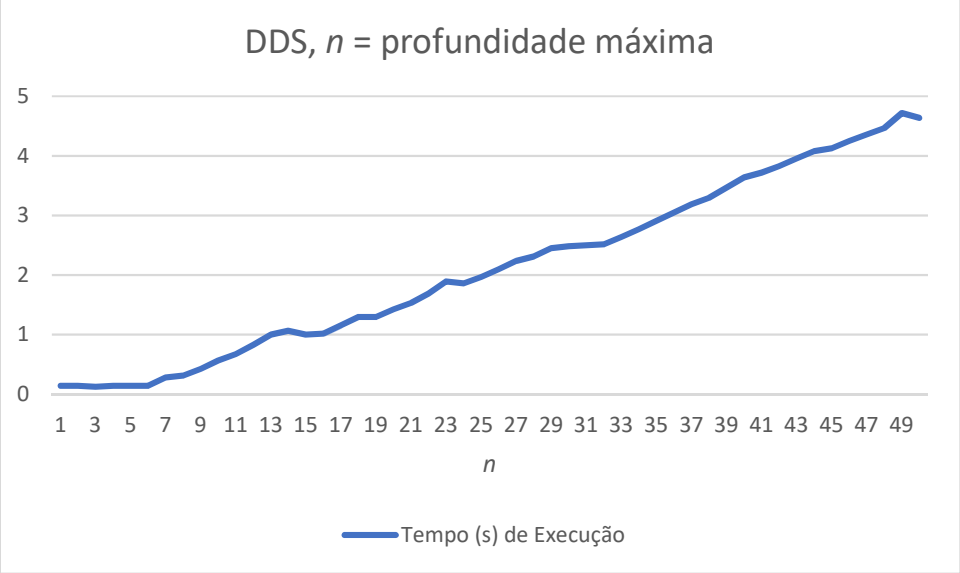
Figura 5. Função auxiliar para o algoritmo DDS

O algoritmo DDS, começa por fazer discrepâncias perto da raiz. Os números de discrepâncias vão aumentando à medida que me afasto da raiz. A heurística tem maior probabilidade de falhar no topo. Em semelhança ao ILDS, compara-se os custos das soluções encontradas entre iterações de forma a encontrar a solução com menor custo. A profundidade actual vai sendo calculada ao longo do algoritmo.

Iterativamente vai aumentando o limite da profundidade. Na iteração 0, vai a favor da heurística. Na iteração i explora os ramos onde as discrepâncias ocorrem na profundidade i ou menos. Tem-se o cuidado de não revisitar nós folha visitados em iteração anteriores. Isto é feito fazendo: na profundidade i da iteração i, vai-se contra a heurística já que ramos que vão a favor da heurística iria nos levar a nós folha visitados anteriormente em iterações anteriores. Em profundidades menores, podemos tomar tanto os ramos a favor como contra heurística. Em profundidades maiores, devemos sempre seguir a heurística. Isto é garantindo pelas verificações feitas da linha 7 à 15 da figura 5.

Foram experimentados vários valores para a profundidade máxima usada no algoritmo, de forma a definir o *n* mais eficiente para encontrar a solução ótima. Em baixo, os gráficos demonstram o comportamento obtido para diferentes valores, para um único teste e heurística *H1*:





É facilmente identificável o melhor valor, n=18. Este valor está fortemente dependente desta procura específica, devido ao funcionamento particular da estratégia. Para problemas maiores, o valor ótimo será maior também – no entanto, a execução do algoritmo deixa de ocorrer em tempo útil. Foi determinado que um problema representativo *pequeno* seria portanto um bom exemplo para calcular o n, e ficou estabelecido n=18.

4. Comparação de resultados

Problema\ Procura	**Profundidade**	**A* (H1)**	**A* (H2)**	**Sondagem Iterativa**	**ILDS**	**DDS**
1	10110	9776	9776	10712	10036	9688
2	21359	21686	21686	23167	21359	21359
3	33305	33536	33536	34470	33228	33305
4	64277	65235	65235	65935	64247	64277
5	93675	91221	91221	93993	93669	93675

Tabela 5 – Custo da solução

Problema\ Procura	**Profundidade**	**A* (H1)**	**A* (H2)**	**Sondagem Iterativa**	**ILDS**	**DDS**
1	0.125	0.156	0.156	1.6875	0.266	1.234
2	1.234375	1.453	1.500	15.1875	2.594	5.109
3	4.234375	6.625	6.719	50.9375	8.750	43.70
4	28.90625	44.83	46.53	354.5625	58.16	172.6
5	85.6875	90.81	92.36	988.5156	170.0	643.8

Tabela 6 – Tempo (s) de execução

Problema\ Procura	**Profundidade**	**A* (H1)**	**A* (H2)**	**Sondagem Iterativa**	**ILDS**	**DDS**
1	82	83	83	820	164	812
2	162	170	170	1620	324	804
3	244	291	291	2440	488	2491
4	472	569	569	4720	944	2932
5	693	710	710	6930	1386	4933

Tabela 7 – Nós expandidos na execução

Problema\ Procura	**Profundidade**	**A* (H1)**	**A* (H2)**	**Sondagem Iterativa**	**ILDS**	**DDS**
1	196	202	202	3053	392	1883
2	436	433	433	6690	872	1965
3	852	861	861	11282	1704	8592
4	1911	1774	1774	25099	3822	11599
5	3679	2822	2822	54913	7358	25843

Tabela 8 – Nós gerados na execução

Para o IDA* não é encontrada uma solução. Isto deve-se ao facto de a estratégia não encontrar um valor *f* maior que o *threshold* inicial: o algoritmo não é adequado a este problema. Já para as estratégias de Profundidade Iterativa e Largura Primeiro, não correm em tempo útil. Em relação à largura, este problema requer uma procura mais afunilada em direção à solução, devido ao tamanho da árvore. A profundidade iterativa tem um problema semelhante, já que a solução deste problema está obrigatoriamente a uma profundidade igual ao número de tarefas (variáveis) inicialmente propostas, e o número de iterações exigidas pelo algoritmo para lá chegar é insustentável.

4.1. Melhor Abordagem

Para problemas pequenos (número de tarefas menor a 100), a melhor abordagem será o DDS. Para problemas médios a grandes (número de tarefas entre 100 e 500), o ILDS comporta-se melhor, e para os maiores problemas (número de tarefas maior a 500), o A* é a estratégia mais eficaz e eficiente.

Em baixo foram registados os resultados obtidos para esta estratégia híbrida, para os mesmos problemas estudados anteriormente:

Problema\ Métricas	**Custo**	**Nº de Turnos**	**Tempo (s)**	**Nós Expandidos**	**Nós Gerados**
1	9688	24	1.25	812	1883
2	21359	51	2.515625	324	872
3	33228	78	8.765625	488	1704
4	64247	151	56.578125	944	3822
5	91221	224	91.046875	710	2822
TOTAL	-	-	*160.15625*	*3278*	*11103*

Comparativamente às outras abordagens, esta estratégia é globalmente melhor (em otimização e eficiência) que todas as outras apresentadas, apesar de sacrificar algum custo computacional em prol de eficácia para problemas mais pequenos. Esta foi a estratégia desenvolvida neste projeto considerada melhor.

4.2. Melhorias a implementar

A função custo podia ser dinâmica. Há medida que o tempo passasse, ia sendo cada vez mais peso até ter o seu peso total. Desta forma, durante a primeira parte da procura o algoritmo mantinha-se com $h(n)>g(n)$, executando mais rapidamente. Durante a outra parte da procura, em que o custo começa a ter o seu peso total, adaptar-se-ia a heurística de forma a que $h(n)<g(n)$, executando a procura mais devagar, mas garantindo um caminho ótimo.

5. Conclusão

Conclui-se que para os problemas correrem em tempo útil, é necessário que a heurística em certa altura seja um pouco maior que a função de custo. No entanto, é necessário ter um equilíbrio entre as partes em que a procura corre com h(n) > g(n), pois se nunca correr com h(n)<g(n), não há garantia que encontre uma solução ótima.

Para problemas pequenos, abordagens que exploram a maior parte do espaço de estados, como no caso dos algoritmos sistemáticos (ILDS e DDS) e a sondagem iterativa, encontram custos menores. Para problemas grandes, o A* comporta-se melhor, visto que não tem em conta só a heurística, como também o custo.

6. Bibliografia

1. Angelo Oddi et al., Iterative-Sampling Search for Job Shop Scheduling with Setup Times, 2009;
2. Richard E. Korf, Improved Limited Discrepancy Search, 1995;
3. Toby Walsh, Depth-Bounded Discrepancy Search, 1996;
4. Patrick Prosser, Chris Unsworth, Limited Discrepancy Search Revisited, 2011.