Assignment 2 Big Triple

Grupo: P2 G4

Trabalho realizado por:

Afonso Guerreiro a75660

Gonçalo Figueiredo a71353

Diogo Pereira a67385

Descrição do problema:

Dado um número inteiro, encontrar a sequência de operações que o transforma no seu triplo, usando um número predefinido de operações, com o menor custo possível. As operações permitidas e os seus custos são os seguintes:

a) Adição de 1 — Custo 1.Exemplo 2 – 3

b) Subtração de 1 — Custo 2. Exemplo 2 – 1

c) Dobro — Custo 3. Exemplo 2 – 4

Descrição dos algoritmos usados:

Para resolver o problema proposto usamos 3 algoritmos de procura state-space. O Uniform cost search, o A* e o IDA*.

Relativamente ao Uniform cost search: O algoritmo serve apenas para inputs de inteiros com valor absoluto pequeno, quando o valor começa a ficar relativamente grande o algoritmo não consegue dar um output em tempo útil. Isto acontece porque o critério de seleção de qual o próximo nó é que se deve expandir é feito apenas com base no custo de ir desde a solução ate o nó em questão, sendo o nó que tiver menor custo o próximo nó a ser expandido. Ora a não utilização de heurísticas neste algoritmo traz grandes desvantagens em comparação com os algoritmos A* e IDA*. Algumas propriedades deste algoritmo são:

- Utiliza priority queue;

- Expande primeiro os nós de menor custo;

- Equivalente à largura-primeiro se os custos forem todos iguais;

- Completo, se custo ≥ ε; ε0;

- Discriminador;

- Complexidade temporal = O(r^ceiling(C*/ ε));

- Complexidade espacial? O(r^ceiling(C*/ ε)); onde C* é o custo da solução ótima;

Propriedades retiradas do pdf: https://tutoria.ualg.pt/2023/pluginfile.php/123768/mod_resource/content/1/ia2023-24T5.pdf

Relativamente ao A* search: Este algoritmo é uma versão "melhorada" do Uniform cost search, pelo menos para este tipo de problemas em que se tem conhecimento de qual é a solução (busca informada), o que vai permitir usar heurísticas, que são estimativas do custo restante para alcançar o objetivo a partir de um determinado nó. Tal como o Uniform cost search este algoritmo usa uma priority queue, mas em vez de se dar prioridade ao no que tiver menor custo, da-se prioridade ao no que tenha um menor valor da função de avaliação: f(n) = g(n) + h(n), onde g(n) é o valor do custo do valor inicial ate ao node n, e h(n) é o custo estimado de ir desde o node n ate a solução. Neste algoritmo é preciso ter atenção para que a heurística seja admissivel:

h^(n) é admissível se para todo o n.

h^(n) = 0 então h^(G)=0 para todo o objetivo G.

Caso a heurística não seja admissível, então A* pode não encontrar uma solução ótima. Algumas propriedades deste algoritmo são:

- Completo

- Discriminador

- Ótimo

- Complexidade temporal: Exponencial

- Complexidade espacial: Exponencial; mantém todos os nós em memória

Propriedades retiradas do pdf:

https://tutoria.ualg.pt/2023/pluginfile.php/131799/mod_resource/content/1/ia2023-24T7.pdf

Relativamente ao IDA* search: Cada iteração deste algoritmo faz uma procura em profundidade para soluções a uma determinada distância, o IDA* search é também mais simples e muitas vezes mais rápido do que A*, tal como no A* a sua rapidez vai depender da qualidade da heurística utilizada na função

de avaliação. Este algoritmo tem também como vantagem em relação ao A* o menor uso de memória. Algumas propriedades deste algoritmo são:

- Completo

- Discriminador

- Ótimo

- Complexidade temporal: Exponencial

- Complexidade espacial: O(b*d), onde b = branching factor (numero de sucessores gerados por um nó) e d = depth ate a solução ótima;

Propriedades retiradas do site:

https://stackoverflow.com/questions/54490981/artificial-intelligence-time-complexity-of-ida-search

Unit tests:

Ver os ficheiros *"TripleTest.java"* e *"BestFirstTest.java"*.

Opções de design:

Para este projeto optamos por usar o design: "Strategy design pattern for State-Space Search" isto porque há inúmeros problemas de State-Space Search e utilizando o Strategy design pattern, em vez de se implementar um algoritmo de procura para um problema em específico, implementa-se um algoritmo de procura geral, onde não é preciso se mexer independentemente do problema a ser resolvido. Isto consegue-se fazer através do uso de uma interface, neste projeto o nome da nossa interface é Ilayout, e se for necessário explorar mais um problema da natureza State-Space Search, basta fazer uma nova classe que implemente o Ilayout e os algoritmos da classe BestFirst, neste caso, vão funcionar para a nova classe.

Código:

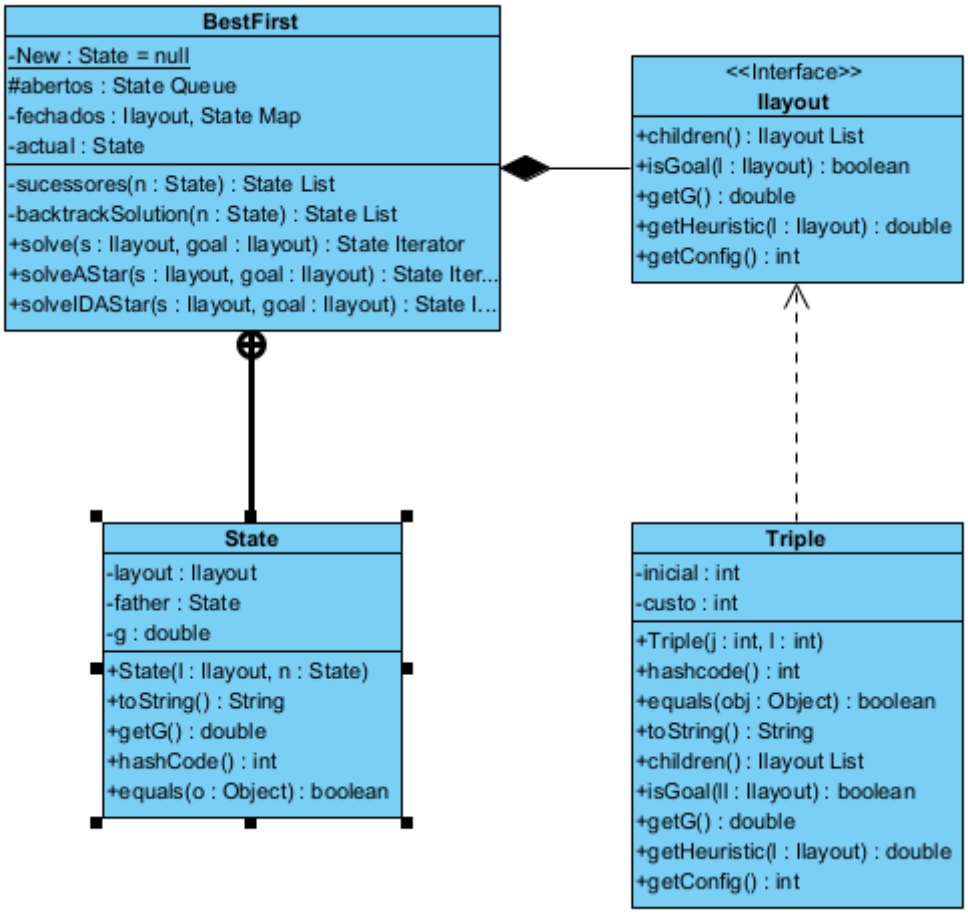
Ver pasta *"src"*.

Javadoc:

Ver pasta *"doc"*.

UML:

Para ver com mais detalhe, ver o ficheiro *"BestTriple.vpp"*



Resultados, análise e discussão:

Os resultados que obtivemos foram de forma geral bons. Para os testes propostos o algoritmo Uniform cost conseguiu encontrar solução em tempo útil para todos os testes com valor absoluto "pequeno", mas nos casos em que os inputs tinham valor absoluto "grande", mais especificamente nos testes para 1000, -2000, 12345 e -12345 o algoritmo não conseguiu encontrar a solução. Devido também a ser uma busca "uninformed" os números de nos expandidos e generados foram muito maiores em comparação com os dos algoritmos A* e IDA*, tendo por consequência também valores de penetrancia muito mais baixos do que os dos outros algoritmos.

Relativamente ao algoritmo A*, é o algoritmo que tem os menores números de nos expandidos e generados, isto por ser um algoritmo de busca informada e por ter um registo de valores já visitados, poupando repeticoes. Por consequência os valores de penetrancia foram os maiores entre os algoritmos estudados, obtendo, por exemplo 0.50 de penetrancia até para um valor grande de input, como o 12345.

Quanto ao IDA*, apesar de ter valores para nos expandidos e generados maiores do que o A*, por não ter nenhum registo de valores já visitados, foi geralmente mais rápido do que o A*. Os valores de penetrancia foram por consequência, menores do que os do A*, mas maiores do que os do IDA*.

Em baixo seguem-se tabelas com o número de nos expandidos, nos generados, distancia da solução e penetrancia, para os 3 algoritmos estudados e com diferentes valores de teste.

E – Numero de nos expandidos

G – Numero de nos generados

L – Distancia da solucao

P – Penetrancia (L / G)

Testes nós expandidos

E	**12345**	**-12345**	**0**	**-23**	**50**	**1**	**6**	**54**	**-50**	**-7**	**1000**	**-2000**
Uniform cost	NA	NA	1	131	3189	4	16	4579	464	35	NA	NA
A*	6175	3098	1	18	27	3	5	29	21	14	502	510
IDA*	24698	55632	1	199	107	10	19	115	280	143	2007	6057

Testes nós generados

G	**12345**	**-12345**	**0**	**-23**	**50**	**1**	**6**	**54**	**-50**	**-7**	**1000**	**-2000**
Uniform cost	NA	NA	0	324	8034	7	35	115556	1168	81	NA	NA
A*	12350	6198	0	39	53	5	9	57	43	31	1003	1021
IDA*	24697	55626	0	193	106	8	18	114	274	138	2006	6054

Testes comprimento da solução

L	12345	-12345	0	-23	50	1	6	54	-50	-7	1000	-2000
Uniform cost	NA	NA	1	10	27	3	5	29	16	6	NA	NA
A*	6175	3090	1	10	27	3	5	29	16	6	502	503
IDA*	6175	3090	1	10	27	3	5	29	16	6	502	503

Testes Penetrancia

P	12345	-12345	0	-23	50	1	6	54	-50	-7	1000	-2000
Uniform cost	NA	NA	∞	0.03	0.00	0.43	0.14	0.00	0.01	0.07	NA	NA
A*	0.50	0.50	∞	0.26	0.51	0.60	0.56	0.51	0.37	0.19	0.50	0.49
IDA*	0.25	0.06	∞	0.05	0.25	0.38	0.28	0.25	0.06	0.04	0.25	0.08

Conclusão e comentários:

Com este trabalho aprimoramos o nosso conhecimento relativamente aos algoritmos Uniform Cost, A* e IDA*, percebendo melhor o seu funcionamento, a influência das heurísticas (se o algoritmo usar) na procura da solução e quais estruturas de dados usadas na implementação dos diferentes algoritmos.

Também é importante acrescentar que inicialmente nos tentamos implementar o algoritmo IDA* recursivamente, só que encontramos problemas quando metíamos inputs de inteiros com valores absolutos muito grandes, obtiamos sempre stackOverflow, por isso vimo-nos obrigados a implementar o IDA* iterativamente.