Inteligência Artificial

Problema 2: Big Triple

Docente: José Valente de Oliveira

Discentes: Afonso Silva nº 76943, Daniel Andrade nº76922

Problema

Dado um inteiro, encontre a sequência de operações que transformam o inteiro inicial no seu triplo, usando um número predefinido de operações, com o menor custo possível.

Só pode ser usado as seguintes três operações com os seguintes custos associados:

- Incremento, i.e., adiciona um ao argumento. Custo: 1

- Decremento, i.e., subtrai um ao argumento. Custo: 2

- Dobro, i.e., multiplica o argumento por 2. Custo: 3

Uniform-cost search

No problema 1 foi utilizado o uniform-cost search, que procura o objetivo pelos nós com menor peso, isto é, o algoritmo prioriza os nós com menor função de avaliação. O algoritmo usa uma Priority-Queue onde os nós com menor função de avaliação para o início da fila, onde a função de avaliação do Uniform-cost search é meramente o peso acumulado até ao nó atual.

Quando um elemento for inserido na *PriorityQueue* este vai percorrer a fila até encontrar um nó com menor avaliação que ele, o elemento é então introduzido nessa posição. Porém, se encontrarmos nós repetidos pode ser um problema, dado que poderíamos percorrer um caminho indesejado, repetido ou até permanecer num ciclo "de trás para frente", isto torna o algoritmo mais lento e pesado, para evitar isto usamos um Hash-Map, onde vamos guardar os nós com os valores que já foram testados.

Input	Tempo médio (Ms)	Memória Média Usada (MB)	Nós Gerados (G)	Nós Expandidos (E)	Tamanho do Caminho (L)	Penetrância (P)
1	0.0222767	0.002656	7	3	2	0.28571428
20	0.05084333	0.109896	362	198	11	0.03038674
40	0.62177	1.05898453	4000	2348	21	0.00525
60	10.098023	5.3436952	38506	23442	31	8.0507E-4
80	155.47819	25.843157	342860	209515	41	1.1958E-4
100	1682.9402	215.11005	2965100	1809062	51	1.7200E-5
120	16788.283	457.40175	25761837	15624534	61	2.3678E-6

Tabela 1. Performance do algoritmo Uniform-cost (Usando o layout *Triple*)

Como podemos ver o crescimento temporal e espacial é exponencial, isto prova que a complexidade temporal e espacial do uniform-cost está certa, sendo esta $O\left(b^{1+[C^*/\varepsilon]}\right)$, onde b é o número de nós expandidos em cada ação, C^* é o custo real, e ε é o menor custo de cada ação.

Em resumo, o algoritmo Uniform-cost usa um como estrutura de dados uma Priority-Queue para guardar os nós que estão para expandir e priorizando os que têm menor custo de avaliação e um Hash-Map para guardarmos os nós testados. O algoritmo começa na raiz expande, procura pelo nó com menor peso acumulado e expande esse pois está no início da fila e a raiz é inserida no mapa, e repetimos o processo até encontrar o objetivo.

Heurística

A uniform-cost search tem um problema, a procura é feita "cegamente", isto é, a uniform-cost prioriza os nós meramente pelo seu peso acumulado, desta forma vamos percorrer diversos caminhos que não são a solução criando nós desnecessários e desperdiçando imenso tempo nessas operações. Precisamos então de alguma forma de saber por onde devemos ir e não basearmo-nos apenas no peso acumulado, aí entra a heurística.

A heurística é um método que estima o peso do nó atual até ao objetivo, este método alavanca muito a eficiência de alguns algoritmos, um exemplo disto é a uniform-cost search que feita com a heurística, esta transforma-se no algoritmo A* que veremos mais tarde.

Se pensarmos bem sobre o problema reparamos que a operação de multiplicação é a mais poderosa multiplicando o valor atual por dois, apenas por custo 3, então o logico é tentar "capitalizar" o máximo possível sobre esta operação. Queremos achar o valor perfeito para multiplicar por 2 $k-1$ vezes. Obtemos k dividindo o objetivo enquanto este for maior que o input. Agora basta "navegar" até $Objetivo\ /\ k-1$ e multiplicá-lo $k-1$ vezes, assim chegamos ao objetivo. Para calcular esse $k-1$ obtemos o seguinte código:

```
function getK(input, objetivo):
        k = 0;
        While |objetivo| > |input| :
                Objetivo /= 2;
                k++;

        return k;
```

Surge um problema com esta abordagem, nem sempre o caminho vai fazer multiplicações, o melhor exemplo disso é quando o input = 1, simplesmente somamos o input três vezes. O caminho com multiplicações teria peso 4, enquanto o caminho só com adições tem peso 3. Agora se juntarmos tudo obtemos uma heurística relativamente boa mas não perfeita, em pseudo código:

```
function distanceFromTo(input, objetivo):
        return objetivo > input ? |objetivo - input| : |objetivo – input| * 2

function heurística(input, objetivo):
        k = getK(input, objetivo)

        valueAbove = objetivo/2^(k - 1)
        valueBelow = objetivo/2^(k)

        sumWeight = distanceFromTo(input, objetivo)
        aboveWeight = distanceFromTo(input, valueAbove) + 3 * (k-1)
        belowWeight = distanceFromTo(input, valueBelow) + 3 * (k)

        return |objetivo| <= |input| ? sumWeight :
        minimo(sumWeight, aboveWeight, belowWeight)
```

Guardamos k numa variável, em seguida criamos duas variáveis, $aboveWeight$ e $belowWeight$, que guardam o valor para o qual o input tem que navegar para multiplicar k ou $k-1$ vezes. Para obtermos o melhor peso, não podemos movermo-nos sempre até $k-1$, existem diversos exemplos

para o qual é melhor voltarmos para trás e multiplicar-mos por 2, por exemplo quando o input: 101 e objetivo: 200, a melhor solução é fazer as seguintes operações, {101 – 1, 100 * 2}, no final o peso será 5, se tivesses usado $k - 1$ o peso seria de 99, porque multiplicar 101 por 2 resultaria num valor acima do nosso objetivo.

Para resolver o problema referido atrás criamos uma variável $sumWeight$ que guarda a distância entre um input e objetivo. No final se o objetivo for menor que o input quer dizer que estamos á frente do objetivo então precisamos voltar para trás, basta retornar a distância entre objetivo e input, se isso não acontecer então retornamos o mínimo entre as 3 variaves que criamos sobre o peso.

Porém a nossa heurística ainda não está admissível, pois no estado atual não aceita números negativos.

```
function distanceFromTo(input, objetivo):
        return objetivo > input ? |objetivo - input| : |objetivo – input| * 2


function heurística(input, objetivo):
        k = getK(input, objetivo)
        negativeOrPositive = objetivo < input ? -0.99 : 0

        valueAbove = objetivo/2^(k - 1) + negativeOrPositive
        valueBelow = objetivo/2^(k) + negativeOrPositive

        sumWeight = distanceFromTo(input, objetivo)
        aboveWeight = distanceFromTo(input, valueAbove) + 3 * (k-1)
        belowWeight = distanceFromTo(input, valueBelow) + 3 * (k)

        return |objetivo| <= |input| ? sumWeight :
        minimo(sumWeight, aboveWeight, belowWeight)
```

Até agora cobrimos tudo o que podíamos par aos números negativos, por isso ser usado o valor absoluto em abundância. Contudo não conseguimos cobrir tudo sobre os números negativos com modulo, isto acontece porque os números negativos vão ter um comportamento diferente mas similar ao mesmo tempo, vejamos então o exemplo do 7 e o seu negativo -7,

7 -> 8 -> 9 -> 10 -> 20 -> 21

-7 -> -6 -> -12 -> -11 -> -22 -> -21

Como podemos ver o comportamento é diferente voltamos para trás em seguida multiplicamos, sucessivamente. Porem, se pensarmos bem é a mesma coisa, o porquê de ser feito desta forma é porque para chegarmos perto de -21 precisamos fazer a operação de subtrair que custa 2, então para evitar usar essa operação o comportamento certo é voltar para trás e multiplicar, o contrário dos números positivos. Até agora as divisões feitas arredondaram para baixo, contudo nos negativos nós queremos ir para cima daí a variável $negativeOrPositive$ ser -0.5 se negativa.

Esta até poderia ser a versão final da nossa heurística, pois apesar de não ser perfeita, esta é admissível pois o peso que retorna não é maior que o peso ideal(C*). Por isso para fazê-la perfeita precisamos de adicionar o resto. Se repararmos no código nos apenas contamos com as operações de navegar até ao valor e depois multiplicá-lo, mas nem todas as divisões por 2 retornam resto 0, então perdemos informação e para recuperá-lo podemos fazer o seguinte,

```
function getRemainder(input, objetivo, k):
        result = 0
        auxObjetivo = |objetivo|

        while k-- > 0:
                result += auxObjetivo  %  2
                auxObjetivo = auxObjetivo/2 + (np + auxObjetivo % 2 / 2)

        return result
```

A função recebe como sempre um input e objetivo, mas desta vez temos um k. O k a ser usado tem que ser o correto, se voltarmos á heurística percebemos que o belowWeight e aboveWeight usam k e $k-1$ respetivamente. Ou seja, usam valores diferentes para os seus cálculos, e ao calcular o resto precisamos saber se fomos “para trás” ou “para a frente”.

O *getRemainder()* “devolve” a informação que perdemos com as divisões, adicionamos a uma variável result, dividimos o objetivo k enquanto adicionamos o resto da divisão por 2 na variável result. Se o valor for negativo e a sua divisão não for inteira nós queremos arredonda-lo para cima, pois como vimos anterior mente nos multiplicamos o valor por 2 e em seguida adicionamos 1 para voltar para trás, esse 1 faz parte do resto da divisão que retornamos na função.

Concluindo então a nossa heurística. Obtemos uma heurística praticamente perfeita, e dizemos isto pois existem dois exemplos para o qual a nossa heurística não cobre, estes são impossíveis de acontecerem por isso decidimos não implementa-los no código para não adicionar complexidade, já que eles não influenciam o resultado final. Os casos são os seguintes, se o input for entre ~ [-3, 3] e o objetivo for maior que o seu triplo, isto nunca acontece pois nunca vamos explorar nós acima do seu triplo e se porventura voltarmos 1 para trás o objetivo ira ser maior que o seu triplo, mas a sua função de avaliação será menor do que a do correto nó a se seguir, e finalmente se o input e o seu objetivo tiverem diferentes sinais.

Ainda assim é preciso ter cuidado pois a heurística não pode superestimar o peso real do nó atual até ao objetivo ou C*, isto deve-se ao facto que podemos nunca encontrar o caminho até ao objetivo, dado que o algoritmo pode "pensar" que o nó atual tem um valor estimado muito alto que escolhe não continuar por esse nó. Por isso a nossa heurística é admivissivel não superestimando o peso ideal.

A* search

A* e Uniform-cost search são equivalentes, mas A* implementaa heuristica, em outras palavras, A* é uma Uniform-cost search com uma função de avaliação:

$$f(n) = g(n) + h(n)$$

onde $g(n)$ é o custo da raiz até ao nó atual, e $h(n)$ é a heurística mais curto de n até o objetivo.

Input	**Tempo médio (Ms)**	**Memória Média Usada (MB)**	**Nós Gerados (G)**	**Nós Expandidos (E)**	**Tamanho do Caminho (L)**	**Penetrância (P)**
1	0.0360266	0.0026560	5	2	2	0.4
20	0.0125533	0.0080000	23	11	11	0.4782608
40	0.0269366	0.0136640	43	21	21	0.4883720
60	0.0445533	0.1197279	63	31	31	0.4920634
80	0.0572333	0.6709066	83	41	41	0.4939759
100	0.07528	1.0485760	103	51	51	0.4951456
120	0.0851040	0.0569363	123	61	61	0.4959349

Tabela 2. Performance do algoritmo A*(Usando o layout *Triple*)

Se compararmos os valores obtidos na **Tabela 1** e **Tabela 2** verificamos o seguinte, no input 1 o uniform-cost é mais rápido, isto pode significar 2 coisas, a heurística abranda o problema quando o caso é muito pequena e são gerados poucos nós, ou o numero é tão baixo que existe muita incerteza sobre o real valor temporal, se virmos o input 60 vemos um crescimento enorme no uniform-cost de tempo e memória enquanto que no A* é um crescimento mais constante, confirmando mais uma vez a exponencialidade do uniform-cost, enquanto que A* é constante como podemos verifica nos seguintes gráficos.

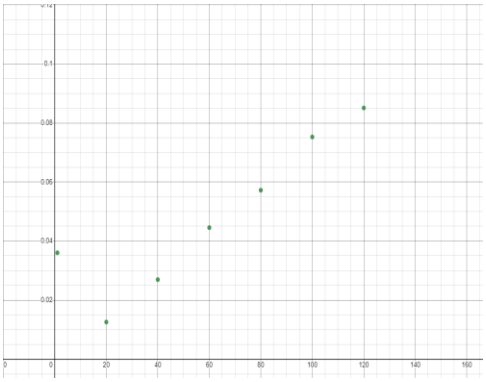


Gráfico 1. A* complexidade temporal

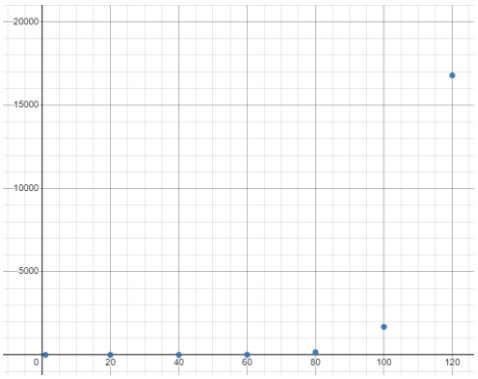


Gráfico 2. Uniform-cost complexidade temporal

A complexidade espacial é idêntica nos dois algoritmos.

A complexidade temporal e espacial no pior dos casos é $O(b^L)$, ou seja, seria exponencial, mas como temos uma heurística, muito boa ou até perfeita, a complexidade temporal é constante.

IDA* search

Se quisermos poupar em memória sacrificando algum tempo, podemos usar o algoritmo IDA* search. Como o nome indica este é uma mistura do Iterative-deepening e A*, usando a vantagem da memória de iterative-deepening e a velocidade do A*.

Diferente do A* no IDA* vamos usar um Stack em vez de uma PriorityQueue, também não vamos verificar os fechados, ou seja, não usamos mais nada sem ser um Stack e a Classe State criado criada por nós.

O algoritmo consiste em termos um limite máximo até onde podemos ir na árvore, se chegarmos até esse limite apagamos a árvore anterior e recomeçamos do zero, isto dá uma vantagem na parte da memória, mas falha no tempo. O limite inicial é a função de avaliação, que é igual á de A*, do primeiro nó, ou seja, o input. Depois apenas expandimos os nós que forem menores ou iguais ao limite, depois de expandirmos colocamos todos os nós no Stack, se chegarmos até ao limite sem acharmos o objetivo recomeçamos do inicio com o limite novo sendo, a menor função de avaliação do menor nó que excedeu o limite.

Input	**Tempo médio (Ms)**	**Memória Média Usada (MB)**	**Nós Gerados (G)**	**Nós Expandidos (E)**	**Tamanho do Caminho (L)**	**Penetrância (P)**
1	0.01621333	0.00265600	6	2	2	0.33333333
20	0.00945333	0.00795200	33	11	11	0.33333333
40	0.01237333	0.01593600	63	21	21	0.33333333
60	0.01508	0.01855200	93	31	31	0.33333333
80	0.02316000	0.02679200	123	41	41	0.33333333
100	0.02764666	0.03264	153	51	51	0.33333333
120	0.08038333	0.04091199	183	61	61	0.33333333

Tabela 3. Performance do algoritmo IDA*(Usando o layout *Triple*)

Igual ao A* o crescimento de IDA* é constante, comparado com uniform-cost, IDA* é absurdamente melhor em termos complexidade temporal e espacial. Sobre os dados da **Tabela 3** podemos ver uma coisa muito interessante, o valor da penetrância é sempre o mesmo, isto só acontece se a heurística for perfeita, pois a função de avaliação do valor inicial vai ser igual ao peso ideal, então só vamos percorrer os nós que têm peso menor que o ideal e como a heurística é perfeita só vai existir 1 que é o nó no caminho correto ou seja nunca verificamos outros nós.

A* vs IDA* search

Tivemos que usar valores muito baixos nas tabelas anteriores devido á ineficiência temporal e espacial do uniform-cost search, então vamos elevar um pouco as coisas para compararmos A* e IDA*.

Input	Tempo médio (Ms)	Memória Média Usada (MB)	Nós Gerados (G)	Nós Expandidos (E)	Tamanho do Caminho (L)	Penetrância (P)
10 000	17.1604633	1.65357227	10003	5001	5001	0.49995002
20 000	33.0763333	3.12289952	20003	10001	10001	0.499975
30 000	57.2141667	4.99807467	30003	15001	15001	0.499983
40 000	75.5842967	6.28307864	40003	20001	20001	0.499987
50 000	96.26983	7.78269467	50003	25001	25001	0.49999
60 000	116.40776	9.09792773	60003	30001	30001	0.49999167
70 000	143.530237	10.5724186	70003	35001	35001	0.499992857

Tabela 4. Performance do algoritmo A* onde input = [10.000, 70.000]

Input	Tempo médio (Ms)	Memória Média Usada (MB)	Nós Gerados (G)	Nós Expandidos (E)	Tamanho do Caminho (L)	Penetrância (P)
10 000	2.33156	2.23018	15003	5001	5001	0.3333333333
20 000	5.36602	1.9702157	30003	10001	10001	0.3333333333
30 000	6.03865567	4.6543216	45003	15001	15001	0.3333333333
40 000	6.18103333	6.2830786	60003	20001	20001	0.3333333333
50 000	10.0360033	7.7657836	75003	25001	25001	0.3333333333
60 000	12.5878599	8.8005122	90003	30001	30001	0.3333333333
70 000	12.4726900	12.8092819	105003	35001	35001	0.3333333333

Tabela 5. Performance do algoritmo IDA* onde input = [10.000, 70.000]

A primeira coisa a reparar é o tamanho do input, que é muito maior que o máximo do uniform-cost. Porém este não é nem de perto o máximo que estes algoritmos podem chegar, ou até poderíamos dizer a heurística, pois é a heurística que puxa os algoritmos a serem tão poderosos, se tivermos uma heurística perfeita o algoritmo simplesmente vai até ao seu objetivo.

A segunda coisa é um pouco estranha, pois dissemos anteriormente que o IDA* perde complexidade temporal para economizar na memória, mas com estes exemplos percebemos que o IDA* é significativamente mais rápido que o A*. Isto deve-se á heurística ser perfeita, o algoritmo nunca vai fazer mais que uma iteração, ou seja nunca vai recomeçar do inicio. Se não temos de recomeçar então agora é fácil perceber o porquê de ser mais rápido, no A* usamos uma

PriorityQueue e no IDA* um Stack, na PriorityQueue vamos ter que percorre-la, já no Stack é dar push.

Outra coisa interessante é que podemos pensar que os nós gerados menos nós no A* do que no IDA*, e é verdade, mas nós podemos reduzir para os nós gerados no IDA* para serem similares com o A*, simplesmente não gerando o nó pai, porque não faz sentido voltarmos para traz, assim em vez de gerarmos 3 nó por nó geramos apenas 2, agora em vez de a penetrância ser 0.3 seria ~0.5, assim como A*.

Input	**Tempo médio (Ms)**	**Memória Média Usada (MB)**
-10 000	5.745313	1.2933736
-20 000	12.750775	2.5496994
-30 000	20.089532	3.862076
-40 000	29.683933	3.1319253
-50 000	39.082012	3.8978615
-60 000	47.27146	5.0094438
-70 000	57.009143	4.2548991

Tabela 6. Performance do algoritmo A* onde input = [-10.000, -70.000]

Input	**Tempo médio (Ms)**	**Memória Média Usada (MB)**
-10 000	0.661971	1.1668016
-20 000	1.30395	2.3013931
-30 000	1.934245	3.4425413
-40 000	3.855396	1.9682838
-50 000	3.24174	5.5876142
-60 000	4.186712	6.1794202
-70 000	4.851194	7.329909

Tabela 7. Performance do algoritmo IDA* onde input = [-10.000, -70.000]

Quando o input é um número negativo, o tempo e execução pode ser até ~3x mais rápido que números positivos, isto deve-se ao facto que o caminho até ao objetivo vai ser sempre menor que os que o caminho dos números positivos. Se compararmos memória fica um pouco difícil pois nos testes que rodamos a memória é sempre inconsistente para números positivos, mas o tempo percebemos que é constante assim como os números positivos.

Conclusão

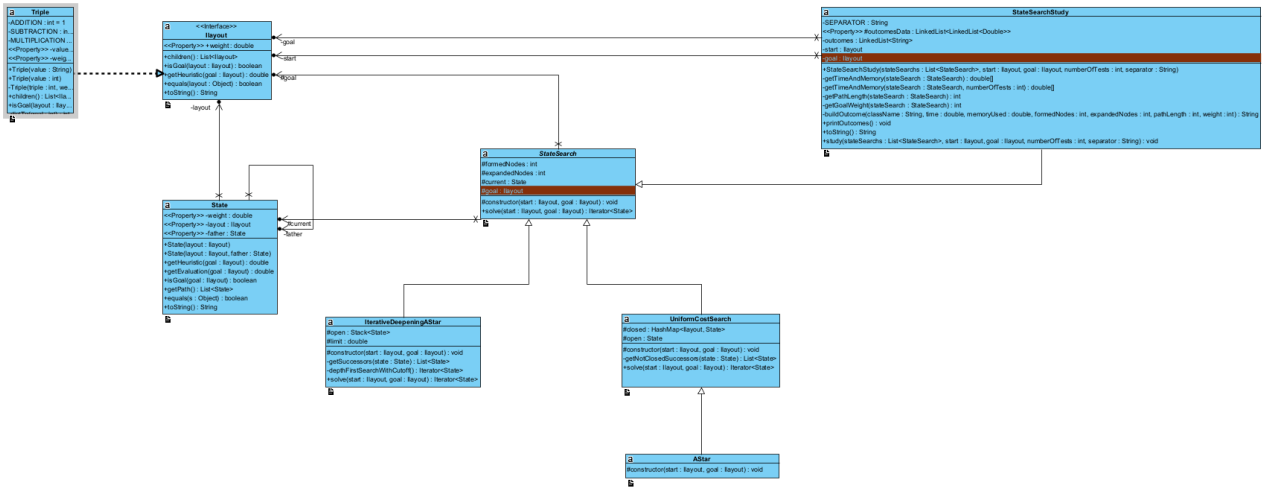
No final, podemos fazer diversas conclusões. O uniform-cost é um algoritmo essencial para podermos para podermos desenvolver diversos outros algoritmos, não é apenas por ser ineficiente que não seja útil para nós.

O uniform-cost pode ser melhorado apenas com um passo que parece muito simples, apenas adicionar uma estimativa na sua função de avaliação, mas dependendo do problema essa estimativa pode ser muito complicada de ser calculada precisamente, ser admissível e ser consistente. Contudo a heurística é um método muito poderoso pois podemos transformar o uniform-cost que neste problema do triplo apenas conseguia gerar exemplos onde o limite do input é 120, enquanto que o seu a sua versão melhorada, A*, consegue chegar ate pelo menos ao input 10,000,000.

Descobrimos que se tivermos uma heurística perfeita, o IDA* torna-se num algoritmo poderoso em termos de complexidade temporal. Pois alteramos a sua natureza de recomeçar, pois nunca remoçamos já que o limite vai ser o custo real e todos os seus sucessores vão ter um peso igual a esse.

Assim, percebemos que o melhor algoritmo a ser usado com a nossa heurística seria, IDA*, mas se a nossa heurística não fosse perfeita, que acontece em problemas mais complexos, então IDA* seria uma boa escolha para complexidade espacial, e A* para complexidade temporal.

UML



Tests

```
class TripleTest {

    @Test
    public void HeuristicTest(){
        Triple tr = new Triple(2);
        Ilayout goal = new Triple(tr.getValue()*3);
        assertEquals(tr.getHeuristic(goal), 4);

        Triple tr1 = new Triple(10);
        Ilayout goal1 = new Triple(tr1.getValue()*3);
        assertEquals(tr1.getHeuristic(goal1), 8);

        Triple tr2 = new Triple(-69);
        Ilayout goal2 = new Triple(-60);
        assertEquals(tr2.getHeuristic(goal2), 9);

        Triple tr3 = new Triple(-420);
        Ilayout goal3 = new Triple(tr3.getValue()*6);
        assertEquals(tr3.getHeuristic(goal3), 114);
    }
}
```

```
class UniformCostSearchTest {
    @Test
    public void UniformCostTests(){
        UniformCostSearch uf = new UniformCostSearch();

        Ilayout st = new Triple(2);
        Ilayout mid = new Triple(3);
        Ilayout goal = new Triple(6);
        Iterator<State> sol = uf.solve(st, goal);
        ArrayList<State> temp = new ArrayList<>();
        temp.add(0, new State(st));
        temp.add(1, new State(mid));
        temp.add(2, new State(goal));
        Iterator<State> test = temp.iterator();
        while(sol.hasNext()){
            assertEquals(sol.next(), test.next());
        }

        Ilayout st1 = new Triple(-2);
        Ilayout mid1 = new Triple(-3);
        Ilayout goal1 = new Triple(-6);
        Iterator<State> sol1 = uf.solve(st1, goal1);
        ArrayList<State> temp1 = new ArrayList<>();
        temp1.add(0, new State(st1));
        temp1.add(1, new State(mid1));
        temp1.add(2, new State(goal1));
        Iterator<State> test1 = temp1.iterator();
        while(sol1.hasNext()){
            assertEquals(sol1.next(), test1.next());
        }

        Ilayout st2 = new Triple(4);
        Ilayout mid2 = new Triple(5);
        Ilayout mid3 = new Triple(6);
        Ilayout goal2 = new Triple(12);
        Iterator<State> sol2 = uf.solve(st2, goal2);
        ArrayList<State> temp2 = new ArrayList<>();
        temp2.add(0, new State(st2));
        temp2.add(1, new State(mid2));
        temp2.add(2, new State(mid3));
        temp2.add(3, new State(goal2));
        Iterator<State> test2 = temp2.iterator();
        while(sol2.hasNext()){
            assertEquals(sol2.next(), test2.next());
        }
    }
```

```
class AStarTest {
    @Test
    public void AStarTests(){
        AStar as = new AStar();

        Ilayout st = new Triple(2);
        Ilayout mid = new Triple(3);
        Ilayout goal = new Triple(6);
        Iterator<State> sol = as.solve(st, goal);
        ArrayList<State> temp = new ArrayList<>();
        temp.add(0, new State(st));
        temp.add(1, new State(mid));
        temp.add(2, new State(goal));
        Iterator<State> test = temp.iterator();
        while(sol.hasNext()){
            assertEquals(sol.next(), test.next());
        }

        Ilayout st1 = new Triple(-2);
        Ilayout mid1 = new Triple(-3);
        Ilayout goal1 = new Triple(-6);
        Iterator<State> sol1 = as.solve(st1, goal1);
        ArrayList<State> temp1 = new ArrayList<>();
        temp1.add(0, new State(st1));
        temp1.add(1, new State(mid1));
        temp1.add(2, new State(goal1));
        Iterator<State> test1 = temp1.iterator();
        while(sol1.hasNext()){
            assertEquals(sol1.next(), test1.next());
        }

        Ilayout st2 = new Triple(4);
        Ilayout mid2 = new Triple(5);
        Ilayout mid3 = new Triple(6);
        Ilayout goal2 = new Triple(12);
        Iterator<State> sol2 = as.solve(st2, goal2);
        ArrayList<State> temp2 = new ArrayList<>();
        temp2.add(0, new State(st2));
        temp2.add(1, new State(mid2));
        temp2.add(2, new State(mid3));
        temp2.add(3, new State(goal2));
        Iterator<State> test2 = temp2.iterator();
        while(sol2.hasNext()){
            assertEquals(sol2.next(), test2.next());
        }
    }
}
```

```
class IterativeDeepeningAStarTest {
    @Test
    public void IDAStarTests(){
        IterativeDeepeningAStar ida = new IterativeDeepeningAStar();

        Ilayout st = new Triple(2);
        Ilayout mid = new Triple(3);
        Ilayout goal = new Triple(6);
        Iterator<State> sol = ida.solve(st, goal);
        ArrayList<State> temp = new ArrayList<>();
        temp.add(0, new State(st));
        temp.add(1, new State(mid));
        temp.add(2, new State(goal));
        Iterator<State> test = temp.iterator();
        while(sol.hasNext()){
            assertEquals(sol.next(), test.next());
        }

        Ilayout st1 = new Triple(-2);
        Ilayout mid1 = new Triple(-3);
        Ilayout goal1 = new Triple(-6);
        Iterator<State> sol1 = ida.solve(st1, goal1);
        ArrayList<State> temp1 = new ArrayList<>();
        temp1.add(0, new State(st1));
        temp1.add(1, new State(mid1));
        temp1.add(2, new State(goal1));
        Iterator<State> test1 = temp1.iterator();
        while(sol1.hasNext()){
            assertEquals(sol1.next(), test1.next());
        }

        Ilayout st2 = new Triple(4);
        Ilayout mid2 = new Triple(5);
        Ilayout mid3 = new Triple(6);
        Ilayout goal2 = new Triple(12);
        Iterator<State> sol2 = ida.solve(st2, goal2);
        ArrayList<State> temp2 = new ArrayList<>();
        temp2.add(0, new State(st2));
        temp2.add(1, new State(mid2));
        temp2.add(2, new State(mid3));
        temp2.add(3, new State(goal2));
        Iterator<State> test2 = temp2.iterator();
        while(sol2.hasNext()){
            assertEquals(sol2.next(), test2.next());
        }
    }
}
```