الغاية من الجلسة:

برمجة خوارزميتي BFS وDFS .

مقدمة:

يمكن تقسيم آلية عمل خوارزميات البحث إلى ثلاث مراحلpick, check, expands فهي تقوم باختيار عقدة ما أولاً تبعا لآلية معينة(تحددها الخوارزمية)ثم تقوم بفحصها فتختبر إن كانت الحالة الهدف وثم تقوم بتوسيعها(إضافة أبناءها) تختلف خوارزميات البحث حسب آلية التوسعة.

نستعمل في خوارزمية BFS رتل يتبع مبدأ FIFO أي من يدخل أولاً يخرج أولاً و بالتالي نضع العقدة على اليمين بينما خوارزمية ال DFS تستعمل مكدس يعمل على مبدأ LIFO أي من يدخل أخيرا يخرج أولاً فنضع العقدة على اليسار.
من المهم معرفة أن خوارزمية ال BFS توجد حل أمثلي في حال كانت كلف الأفعال موحدة ,وخوارزمية DFS في حال الفضاءات غير المنتهية قد تدخل في حلقة لا تستطيع الخروج منها.
هذه الخوارزميات تتعامل مع أغراض من الصف Node فيلزمنا تحويل الحالة إلى Nodeحيث تملك Node متحولا يرتبط بالعقدة الأب و متحولا يرتبط بالفعل الذي طبقناه لنصل لهذه العقدة(أي الوصلة التي عبرناها)و بجلب العقدة الأب و الأفعال للعقدة الهدف مرة تلو الأخرى(والتي تردها خوارزميات البحث)سنصل للجذر وبالتالي نحصل على طريق من الحالة البدائية للحالة النهائية هو الحل ,أيضا صف العقدة يملك متحولا يرتبط بكلفة الطريق الذي يؤدي للعقدة.
سنقوم الآن بإنشاء حزمة Algorithms داخل الملف core ونضيف عليها صف هو BFSوهناك خطوات ثابتة فأي خوارزمية سنقوم ببرمجتها ترث الصف SearchAlgorithmوتحتاج لاستيراد الآتي *.core و *.java.util كما أننا يجب أن نعيد تعريف التابع search والذي نمرر له وسيط من صف المسائلProblem ويرد لنا Node هي العقدة الهدف:

```
package core.algorithms;
import core.*;
import java.util.*;
public class BFS extends SearchAlgorithm {
@Override
        public Node search(Problem p) {}
}
```

الآن كما قلنا سابقا لتطبيق خوارزمية ال BFS نحتاج لرتل نخزن فيه العقد Nodes التي سنقوم بفحصها ومصفوفة نخزن فيها الحالات State التي قمنا باكتشافها مسبقا ومصفوفة نخزن فيها الحالات States التي يمكن الوصول لها من العقدة الحالية (الحالات الأبناء)يعود السبب في كونها مصفوفة حالات إلى أن صف المسألة يملك حالات وليس عقد فعند إضافة الحالات نقوم بتحويلها لعقد لكي نستطيع تطبيق الخوارزميات عليها)وأيضا نحتاج متحول لتحديد ترتيب العقد المُزارة(مثلا العقدة الجذر رقمها 0 ثم ابناءها .... 2 1 وهكذا).

```
package core.algorithms;
import core.*;
import java.util.*;
public class BFS extends SearchAlgorithm {
	private Queue<Node> container = new
LinkedList<Node>();
	private List<State> explored = new ArrayList<>();
	private Collection<State> nextStates = new ArrayList<>();
	private int visitingOrder = 0;
	public BFS(){}
	public BFS(boolean useGraphSearch) {
		this.useGraphSeach = useGraphSearch;
@Override
	public Node search(Problem p) {} }
```

قمنا بتعريف رتل من العقد Node سميناه Container هو عبارة عن LinkedListمن العقد(الصف Queue صف مجرد لذلك نستعملLinkedList) وعرفنا لائحة List من الحالات States أسميناه explored تمثل العقد المستكشفة وهي عبارة عن ArrayList من الحالات(الصف List صف مجرد لذلك نستعمل) ArrayListبشكل مشابه عرفنا مجموعة من الحالات nextStates نضع بها الحالات التي سنقوم بتحويلها لعقد لكي نضيفها على الرتل، وعرفنا متحول visitingOrderهيأناه بصفر يمثل ترتيب اكتشاف العقد.
الآن سنعرف التابع search والذي نمرر له وسيط هو المسألة و يرد لي العقدة الهدف:

```
package core.algorithms;
import core.*;
import java.util.*;
public class BFS extends SearchAlgorithm {
	private Queue<Node> container = new
LinkedList<Node>();
	private List<State> explored = new ArrayList<>();
	private Collection<State> nextStates = new ArrayList<>();
	private int visitingOrder = 0;
	public BFS(){}
	public BFS(boolean useGraphSearch) {
```

```
        this.useGraphSeach = useGraphSearch;}
@Override
    public Node search(Problem p) {
        Node startNode = new Node (p.getInitialState());
```

نقوم هنا بجلب عقدة البداية للمسألة p ونمررها لباني العقدة فنحولها لعقدة اسمها Start Nodeتمثل عقدة البداية.

```
        container.add(startNode);
```

نضيف عقدة البداية للرتل

```
        while (!(container.isEmpty())) {
```

نقوم بالتكرار طالما أن الرتل لم يصبح فارغا

```
            Node currentNode = container.poll();
```

نسحب أول عقدة من الرتل container عبر التعليمة poll ونهيئ العقدة CurrentNode

```
            currentNode.setVisitingOrder(++visitingOrder);
```

نقوم بتهيئة ترتيب زيارة العقدة الحالية عبر زيادة المتحول visitingOrder بمقدار 1

```
            if (useGraphSeach &&
explored.contains(currentNode.getState())) {
```

في حال كان نمط البحث هو بحث بياني useGraphSeach والمجموعة المستكشفة تحوي حالة العقدة الحالية(أي أن العقدة مكررة والبحث البياني كما نعلم لا يسمح بالتكرار)

```
                continue;
```

انتقل للعقدة التالية في الرتل.

```
}
            if (p.isGoal(currentNode.getState())) { return
currentNode;}
```

نجلب حالة العقدة الحالية currentNode.getState() ونختبر إن كانت هذه الحالة

هي حالة هدف للمسألة p عبر التابعisGoal

```
explored.add(currentNode.getState());
```

ضف حالة العقدة الحالية للمجموعة. Explored

```
for (IAction action : p.getActions()) {
```

قم بجلب كل الأفعال(الوصلات)الممكنة في المسألة p و من أجل كل فعل

```
nextStates = action.apply(currentNode.getState());
```

طبق الفعل على حالة العقدة الحالية وضع العقد الناتجة فيnextStates

```
for (State s : nextStates) {
```

من أجل كل حالة في المصفوفةNextStates

```
double actionCost = p.getActionCost(currentNode.getState(),
action, s);
```

نقوم بحساب كلفة الفعل عبر تابع حساب التكلفة للمسألةp.getActionCost

```
double pathCost =
 actionCost + currentNode.getPathCost();
```

نجمع كلفة الانتقال للعقدة ابن مع كلفة العقدة الحالة و نضعها بمتحول هوpathCost

```
Node child = new Node(s, currentNode, action, pathCost);
```

نقوم بتحويل الحالة لعقدة عبر تمريرها كوسيط ثم تمررين العقد الأب CurrentNodeو الفعل(الوصلة التي عبرناها لنصل لها action)وكلفة الطريق pathCostبهذا الترتيب.

```
container.add(child);
```

قم بإضافة العقدة الابن للرتل

```
}
}
}
```

في حال لم نجد العقدة الهدف لا نرد شيئاreturn null;

```
}
}
```

أي أننا كل مرة سنستمر بفحص العقدة الحالية وجلب الحالات الأبناء التي يمكن الانتقال لها عبر الأفعال ثم تحويل هذه الحالات لعقد و اختبار إن كانت حالات هذه العقد هي حالة هدف ثم إضافتها للرتل لكي يتم توليد أبنائها لاحقا حتى يصبح الرتل فارغا فإن لم نجد عقدة هدف لن نرد شيئا خوارزمية ال DFS هي نفسها تماما و لكن الفرق الوحيد أنها تستعمل مكدس Stackوتستعمل التعليمة pop بدلاً من poll لسحب العقدة و التعليمة push بدلاً من addلإضافة عقدة سنشير لهذه التعليمات بلون غامق ولكن الكود هو نفسه بالعموم

```
package core.algorithms;
import core.*;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.Stack;
public class DFS extends SearchAlgorithm {
private Stack<Node> container = new Stack<>();
        private List<State> explored = new ArrayList<>();
        private Collection<State> nextStates = new
ArrayList<>();
        private int visitingOrder = 0;
        public DFS() {}
        public DFS(boolean useGraphSearch) {
                this.useGraphSeach = useGraphSearch; }
@Override
       public Node search(Problem p) {
               Node startNode = new Node(p.getInitialState());
//create start node
               container.add(startNode);
               while (!(container.isEmpty())) {
                     Node currentNode = container.pop();
                     currentNode.setVisitingOrder(++visitingOrder);
                     if (useGraphSeach &&
explored.contains(currentNode.getState())) {
                                  continue;
}
                      if (p.isGoal(currentNode.getState())) {
                                  return currentNode;
}
                explored.add(currentNode.getState());
                for (IAction action : p.getActions()) {
                        nextStates =
action.apply(currentNode.getState());
                for (State s : nextStates)
```

```
double actionCost = p.getActionCost(currentNode.getState(),
action, s);

double pathCost = actionCost + currentNode.getPathCost();
Node child = new Node(s, currentNode, action, pathCost);
container.push(child)
}
}
}
      return null;}}
```