2. خوارزميات الذكاء الاصطناعي

سيتعرف الطالب في هذه الوحدة على بعض الخوارزميات الأساسية المُستخدَمة في الذكاء الاصطناعي (AI). كما سيتعلم كيف يُنْشِئ نظام تشخيص طبي بسيط مُستنِد إلى القواعد بِطُرق برمجية مُتعددة ثم يقارن النتائج. وفي الختام سيتعلّم خوارزميات البحث وطرق حل ألغاز المتاهة مع أخذ معايير معيّنة في الاعتبار.

أهداف التعلُّم

بنهاية هذه الوحدة سيكون الطالب قادرًا على أن:

- يُنشئ مقطعًا برمجيًّا تكراريًّا.
- يُقارِن بين خوارزمية البحث بأولوية الاتساع وخوارزمية البحث بأولوية العُمق.
- يَصِف خوارزميات البحث وتطبيقاتها.
- يُقارِن بين خوارزميات البحث.
- يَصِف النظام القائم على القواعد.
- يُدرِّب نماذج الذكاء الاصطناعي حتى تتعلّم حل المشكلات المُعقدة.
- يُقيِّم نتائج المقطع البرمجي وكفاءة البرنامج الذي أنشأه.
- يُطوِّر البرامج لمحاكاة حلّ مشكلات الحياة الواقعية.
- يُقارِن بين خوارزميات البحث.

الأدوات

- مفكرة جوبيتر (Jupyter Notebook)

رابط الدرس الرقمي

www.ien.edu.sa

الدرس الأول

الاستدعاء الذاتي

تقسيم المُشكلة Dividing the Problem

في هذا الدرس، ستتعلّم استخدام الدوال التكرارية لتبسيط البرنامج وزيادة كفاءته.

تخيّل أن والدكَ قد أحضرا لكَ هديّة، وكنت مُتلهفًا لمعرفتها، ولكن عندما فتحت الصندوق، وجدت صندوقًا جديدًا بداخله، وعندما فتحته، وجدتَ آخرَ بداخله، وهكذا حتى عجزت أن تعرف في أي صندوق توجد الهدية.

الاستدعاء الذاتي Recursion

الاستدعاء الذاتي هو أحد طُرُق حل المشكلات في علوم الحاسب، ويتم عن طريق تقسيم المشكلة إلى مجموعة من المشكلات الصغيرة المُشابهة للمشكلة الأصلية حتى يُمكنك استخدام الخوارزمية نفسها لحل تلك المشكلات. يُستخدَم الاستدعاء الذاتي بواسطة أنظمة التشغيل والتطبيقات الأخرى، كما تدعمه معظم لغات البرمجة.

يحدث الاستدعاء الذاتي عندما تتكرر التعليمات نفسها، ولكن مع بيانات مختلفة وأقل تعقيدًا.

شكل 2.1: مثال على الاستدعاء الذاتي

لتُلقِ نظرة على مثال لدالة تستدعي دالة أخرى.

```
def mySumGrade (gradesList):
    sumGrade=0
    l=len(gradesList)
    for i in range(l):
        sumGrade=sumGrade+gradesList[i]
    return sumGrade

def avgFunc (gradesList):
    s=mySumGrade(gradesList)
    l=len(gradesList)
    avg=s/l
    return avg

# program section
grades=[89,88,98,95]
averageGrade=avgFunc(grades)
print ("The average grade is: ",averageGrade)
```

استدعاء الدالة mySumGrade.

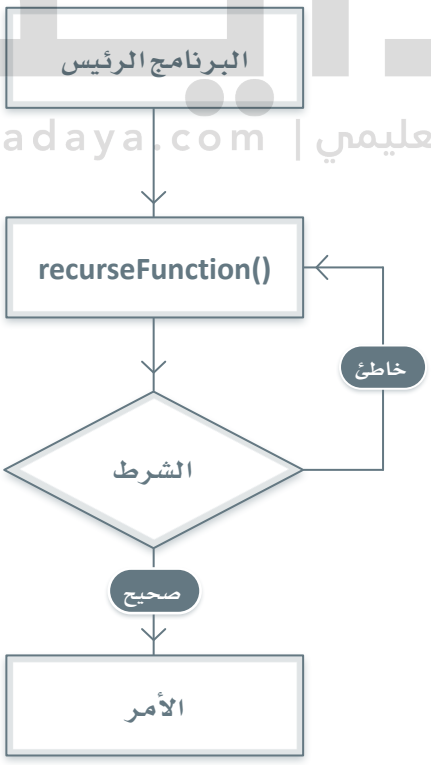
تستخدم دالة len() قائمة كمُعامِل مُدخَل، لحساب وتحديد عدد العناصر في القائمة.

```
The average grade is: 92.5
```

دالة الاستدعاء التكرارية Recursive Function

في بعض الحالات تستدعي الدّالةُ نفسَها وهذه الخاصية تُسمى الاستدعاءات التكرارية (Recursive Calls).

يكون بِناء الجملة العام لدالة الاستدعاء التكرارية على النحو التالي:



شكل 2.2: تمثيل الاستدعاء التكراري

```
# recursive function
def recurseFunction():
    if (condition): # base case
        statement
    else:
        #recursive call
        recurseFunction()

# main program
.......

# normal function call
recurseFunction()
.........
```

الاستدعاء التكراري هو عملية استدعاء الدالة لنفسها.

تتكون دالة الاستدعاء التكرارية من حالتين:

الحالة الأساسية Base Case

وفي هذه الحالة تتوقف الدالة عن استدعاء نفسها، ويتأكّد الوصول إلى هذه الحالة من خلال الأمر المشروط. بدون الحالة الأساسية، ستَتَكرَّر عملية الاستدعاء الذاتي إلى ما لا نهاية.

حالة الاستدعاء التكرارية Recursive Case

وفي هذه الحالة تستدعي الدالةُ نفسَها عندما لا تُحقق شرط التوقف، وتظل الدالة في حالة الاستدعاء الذاتي حتى تصل إلى الحالة الأساسية.

أمثلة شائعة على الاستدعاء الذاتي Recursion Common Examples

أحد الأمثلة الأكثر شيوعًا على استخدام الاستدعاء الذاتي هو عملية حساب مضروب رقم مُعيّن. مضروب الرقم هو ناتج ضرب جميع الأعداد الطبيعية الأقل من أو تساوي ذلك الرقم. يُعبَّر عن المضروب بالرقم متبوعًا بالعلامة "!"، على سبيل المثال، مضروب الرقم 5 هو !5 ويساوي 5*4*3*2*1.

جدول 2.1: مضروب الأرقام من 0 إلى 5

0!	0!=1	
1!	1!=1*1=1	أو 1!= 0! *1
2!	2!=2*1=2	أو 2!= 1! *2
3!	3!=3*2*1=6	أو 3!= 2! *3
4!	4!=4*3*2*1=24	أو 4!=3! * 4
5!	5!=5*4*3*2*1=120	أو 5!=4! * 5

ستلاحظ أن عملية حساب المضروب تستند إلى القاعدة أدناه:

$$n! = \begin{cases} 1 & if\ n=0 \\ (n-1)! * n & if\ n>0 \end{cases}$$

الحالة الأساسية

حالة الاستدعاء التكرارية

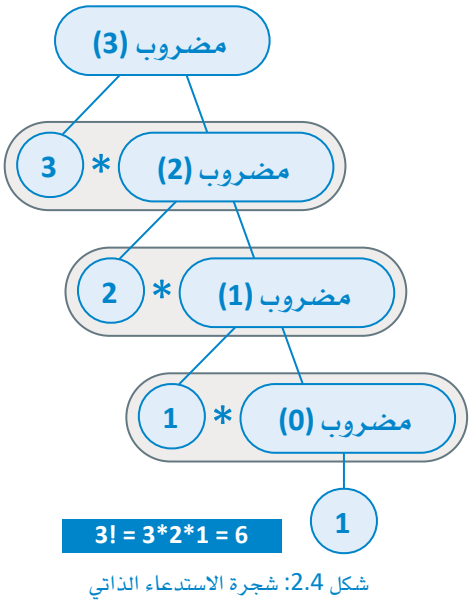
شكل 2.3: قاعدة حساب المضروب

لإنشاء برنامج يقوم باحتساب مضروب العدد باستخدام حلقة التكرار for، اتّبع ما يلي:

```
# calculate the factorial of an integer using iteration

def factorialLoop(n):
    result = 1
    for i in range(2,n+1):
            result = result * i

    return result

# main program
num = int(input("Type a number: "))
f=factorialLoop(num)
print("The factorial of ", num, " is:", f)
```

```
Type a number: 3
The factorial of 3 is:6
```

الآن احسِب مضروب العدد باستخدام دالة المضروب.



شكل 2.4: شجرة الاستدعاء الذاتي

```
# calculate the factorial of an integer using a
# recursive function
def factorial(x):
    if x == 0:
        return 1
    else:
        return (x * factorial(x-1))

# main program
num = int(input("Type a number: "))
f=factorial(num)
print("The factorial of ", num, " is: ", f)
```

الحالة الأساسية.

حالة الاستدعاء التكرارية.

```
Type a number: 3
The factorial of 3 is: 6
```

جدول 2.2: مزايا الاستدعاء الذّاتي، وعيوبه

المزايا	العيوب
• تقلل دوال الاستدعاء التكرارية من عدد التعليمات في المقطع البرمجي. • يمكن تقسيم المهمة إلى مجموعة من المشكلات الفرعية باستخدام الاستدعاء الذاتي. • في بعض الأحيان، يَسهُل استخدام الاستدعاء الذاتي لاستبدال التكرارات المُتداخلة.	• في بعض الأحيان، يَصعُب تَتبُّع منطق دوال الاستدعاء التكرارية. • يتطلب الاستدعاء الذاتي مزيدًا من الذاكرة والوقت. • لا يسهل تحديد الحالات التي يمكن فيها استخدام دوال الاستدعاء التكرارية.

الاستدعاء الذاتي والتكرار Recursion and Iteration

يُستخدم كلٌّ من الاستدعاء الذاتي والتكرار في تنفيذ مجموعة من التعليمات لعدة مرات، والفارق الرئيس بين الاستدعاء الذاتي والتكرار هو طريقة إنهاء الدالة التكرارية. دالة الاستدعاء التكرارية تستدعي نفسها وتُنهي التنفيذ عندما تصل إلى الحالة الأساسية. أما التكرار فيُنفِّذ لِبنَةَ المقطع البرمجي باستمرار حتى يتحقق شرط مُحدَّد أو ينقضي عدد مُحدَّد من التكرارات.

الجدول التالي يعرض بعض الاختلافات بين الاستدعاء الذاتي والتكرار.

جدول 2.3: التكرار والاستدعاء الذاتي

التكرار	الاستدعاء الذاتي
سريع التنفيذ.	بطيء التنفيذ مقارنةً بالتكرار.
يتطلب حجم ذاكرة أقل.	يتطلب حجم ذاكرة أكبر.
حجم المقطع البرمجي أكبر.	حجم المقطع البرمجي أصغر.
ينتهي باستكمال العدد المُحدَّد من التكرارات أو تحقيق شرط مُعيَّن.	ينتهي بمجرد الوصول إلى الحالة الأساسية.

متى تَستخدِم الاستدعاء الذاتي؟

- يُعدُّ الاستدعاء الذاتي الطريقة الأكثر ملائمة للتعامل مع المشكلة في العديد من الحالات.
- يَسهُل استكشاف بعض هياكل البيانات باستخدام الاستدعاء الذاتي.
- بعض **خوارزميات التصنيف** (Sorting Algorithms)، تَستخدِم الاستدعاء الذاتي، مثل: **التصنيف السريع** (Quick Sort).

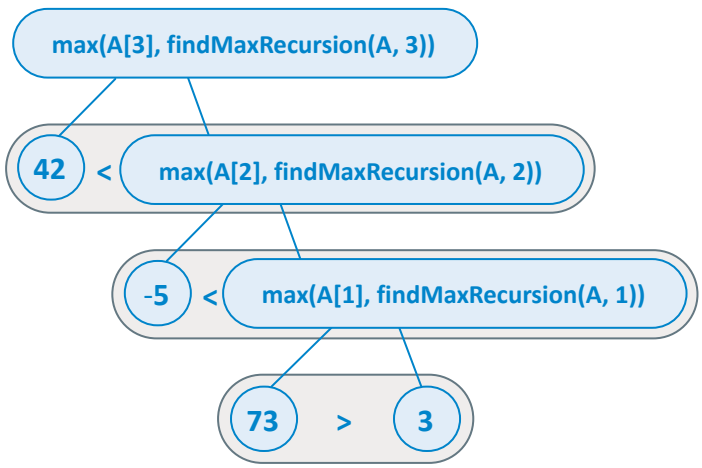
في المثال التالي، ستستخرج أكبر رقم موجود في قائمة مكونة من الأرقام باستخدام دالة الاستدعاء التكرارية.

كما يظهر في السطر الأخير من المثال دالة أخرى للتكرار لغرض المقارنة.

```
def findMaxRecursion(A,n):

    if n==1:
        m = A[n-1]
    else:
        m = max(A[n-1],findMaxRecursion(A,n-1))
    return m

def findMaxIteration(A,n):

    m = A[0]
    for i in range(1,n):
        m = max(m,A[i])
    return m

# main program
myList = [3,73,-5,42]
l = len(myList)
myMaxRecursion = findMaxRecursion(myList,l)
print("Max with recursion is: ", myMaxRecursion)
myMaxIteration = findMaxIteration(myList,l)
print("Max with iteration is: ", myMaxIteration)
```

تستخرج الدالة ()max العنصر ذا القيمة الأكبر (العنصر ذو القيمة الأكبر في myList).

```
Max with recursion is:  73
Max with iteration is:  73
```



شكل 2.5: شجرة الاستدعاء الذاتي لدالة استخراج أكبر رقم في قائمة مكونة من الأرقام

في البرنامج التالي، ستُنشئ دالة استدعاء تكرارية لحساب مُضاعَف الرقم.

ستقوم بإدخال رقمًا (الأساس) وفهرسًا (الأُس أو القُوَّة) يقبلهما البرنامج، ومن ثَمَّ ستَستخدِم دالة الاستدعاء التكرارية powerFunRecursive( ) التي ستَستخدِم هذين المدخَلَين لحساب مُضاعَف الرقم. يمكن تحقيق الأمر نفسه باستخدام التكرار، والمثال التالي يوضّح ذلك:

```
def powerFunRecursive(baseNum,expNum):
    if(expNum==1):
       return(baseNum)
    else:
       return(baseNum*powerFunRecursive(baseNum,expNum-1))

def powerFunIteration(baseNum,expNum):

    numPower = 1
    for i in range(exp):
        numPower = numPower*base
    return numPower

# main program
base = int(input("Enter number: "))
exp = int(input("Enter exponent: "))
numPowerRecursion = powerFunRecursive(base,exp)
print( "Recursion: ", base, " raised to ", exp, " = ",numPowerRecursion)
numPowerIteration = powerFunIteration(base,exp)
print( "Iteration: ", base, " raised to ", exp, " = ",numPowerIteration)
```

```
Enter number: 10
Enter exponent: 3
Recursion: 10 raised to 3 = 1000
Iteration: 10 raised to 3 = 1000
```

دالة الاستدعاء التكرارية اللانهائية Infinite Recursive Function

يجب أن تكون حذرًا للغاية عند تنفيذ الاستدعاء التكراري، كما يجب عليك استخدام طريقة معينة لإيقاف التكرار عند تحقيق شرط مُحدَّد لتجنب حدوث الاستدعاء التكراريّ اللانهائيّ، الذي يسبّب توقُّف النظام عن الاستجابة بسبب كثرة استدعاءات الدالة، مما يؤدي إلى **فَيْض الذاكرة** (Memory Overflow) وإنهاء التطبيق.

تمرينات

1

حدِّد الجملة الصحيحة والجملة الخاطئة فيما يلي:	صحيحة	خاطئة
1. تتكون دالة الاستدعاء التكرارية من حالتين.	✓	○
2. تستدعي دالة الاستدعاء التكرارية دالة أخرى.	○	✓
3. دوال الاستدعاء التكرارية أسرع في التنفيذ.	✓	○
4. استدعاء الدوال يجعل لبِنة المقطع البرمجي أصغر حجمًا.	✓	○
5. كتابة مقطع برمجي مُتكرِّر يتطلب استدعاءً ذاتيًّا أقل.	○	✓

دالة الاستدعاء التكرارية تستدعي نفسها

أكثر

2 ما الاختلافات بين التكرار والاستدعاء الذاتي؟

التكرار	الاستدعاء الذاتي
سريع التنفيذ.	بطيء التنفيذ مقارنة بالتكرار .
يتطلب حجم ذاكرة أقل.	يتطلب حجم ذاكرة أكبر .
حجم المقطع البرمجي أكبر .	حجم المقطع البرمجي أصغر .
ينتهي باستكمال العدد المحدد من التكرارات أو تحقيق شرط معين .	ينتهي بمجرد الوصول إلى الحالة الأساسية .

3 متى يجب استخدام الاستدعاء الذاتي؟

- يعد الاستدعاء الذاتي الطريقة الأكثر ملاءمة للتعامل مع المشكلة في العديد من الحالات
- يسهل استكشاف بعض هياكل البيانات باستخدام الاستدعاء الذاتي
- بعض خوارزميات التصنيف تستخدم الاستدعاء الذاتي

4 وَضِّح مزايا استخدام الاستدعاء الذاتي وعيوبه.

المزايا : - تقلل دوال الاستدعاء التكرارية من عدد من التعليمات في المقطع البرمجي.
- يمكن تقسيم المهمة إلى مجموعة من المشكلات الفرعية باستخدام الاستدعاء الذتي.
- في بعض الأحيان ،يسهل استخدام الاستدعاء الذاتي لاستبدال التكرارات المتداخلة.

المساوئ: - في بعض الأحيان ،يصعب تتبع منطق دوال الاستدعاء التكرارية.
- يتطلب الاستدعاء الذاتي مزيداً من الذاكرة والوقت .
- لا يسهل تحديد الحالات التي يمكن فيها استخدام دوال الاستدعاء التكرارية .

5 اكتب دالة استدعاء تكرارية بلغة البايثون تقوم بحساب الرقم الأكبر بترتيب محدد (مثلًا ثاني أكبر رقم) في قائمة من الأرقام.

6 اكتُب دالة استدعاء تكرارية بلغة البايثون لحساب مجموع كل الأرقام الزوجية في قائمة معيّنة.

رابط الدرس الرقمي

www.ien.edu.sa

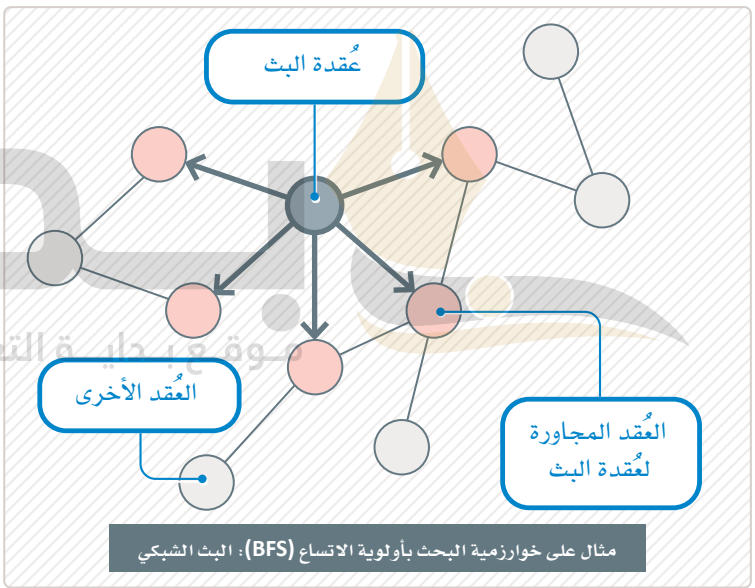
الدرس الثاني

خوارزمية البحث بأولوية العمق والبحث بأولوية الاتساع

البحث في المُخطَّطات Searching in Graphs

هناك بعض الحالات التي تحتاج فيها إلى البحث عن عُقدة مُحدَّدة في المُخطَّط، أو تفحُّص كل عُقدة في المُخطَّط لإجراء عملية بعينها مثل طباعة عُقد المُخطَّط، فتكون حالتكَ كشخصٍ يبحث عن المدينة التي يريد السّفر إليها؛ و ليتحقق هذا، تحتاج إلى فحص كل عُقدة في المُخطَّط حتى تجد تلك التي تحتاج إليها. يُطلق على هذا الإجراء: البحث في المُخطَّط أو مسح المُخطَّط، وهناك العديد من خوارزميات البحث التي تساعد على تنفيذه، مثل:

- **خوارزمية البحث بأولوية الاتساع** (Breadth-First Search - BFS).
- **خوارزمية البحث بأولوية العمق** (Depth-First Search - DFS).



مثال على خوارزمية البحث بأولوية الاتساع (BFS): البث الشبكي

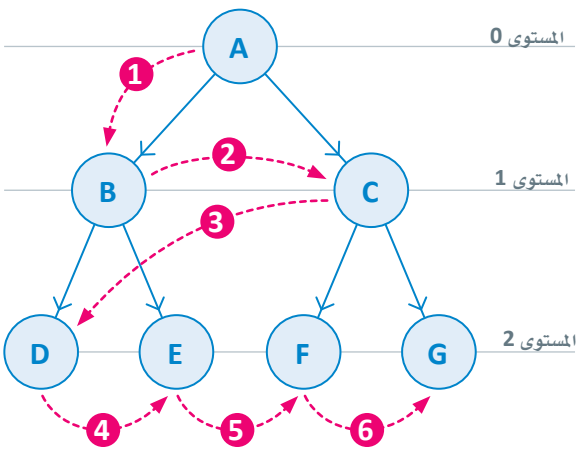
مثال على خوارزمية البحث بأولوية العمق (DFS): حل المتاهة

خوارزمية البحث بأولوية الاتساع
Breadth-First Search (BFS) Algorithm

تستكشف خوارزمية **البحث بأولوية الاتساع** (BFS) المُخطَّط بحسب المستوى واحدًا تلو الآخر، حيث تبدأ بفحص عُقدة الجذر (عُقدة البداية)، ثم تفحص جميع العُقد المرتبطة بها بشكل مباشر واحدة تلو الأخرى.

بعد الانتهاء من فحص كل العُقد في المستوى، تنتقل إلى المستوى التالي، وتتبع الإجراءات نفسها المُوضّحة في الشكل 2.6.

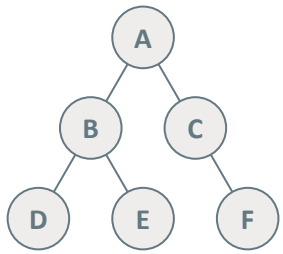
يُستخدَم الطّابور لتتبّع العُقد التي تمّ فحصها، وبمجرّد استكشاف العُقدة، ستتم إضافة العُقَد الفرعية إلى الطابور، ثم تحذف العُقدة التالية الموجودة في أول الطابور التي تم استكشافها سابقًا.



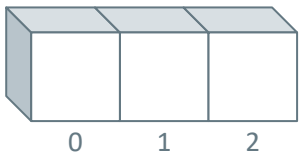
شكل 2.6: خوارزمية البحث بأولوية الاتساع (BFS)

المثال التالي يوضح طريقة عمل خوارزمية **البحث بأولوية الاتساع** (BFS). باستخدام المُخطَّط التالي، حدِّد العُقد التي يجب فحصها للانتقال من عُقدة الجذر A إلى العُقدة F:

ملاحظة: استخدِم هيكل البيانات المُناسب.



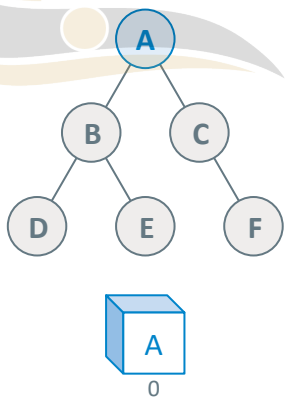
المُخطَّط



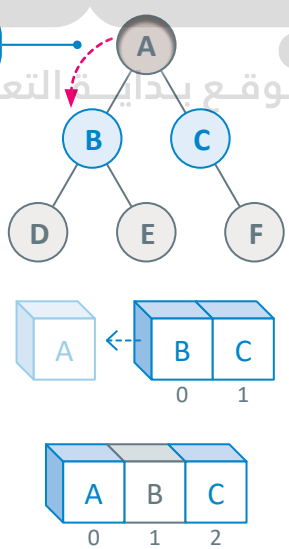
الطابور

عليك فحص كل العُقد في المستوى 1 قبل الانتقال إلى العُقد في المستوى 2.

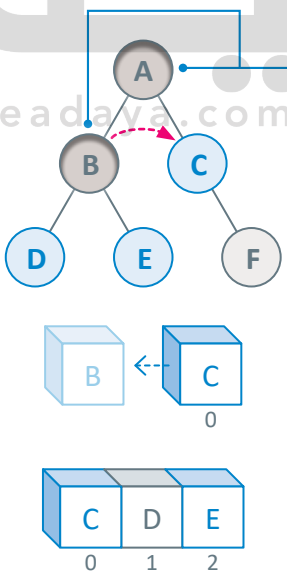
1 البداية من العُقدة الجذرية (العُقدة A). أضف العُقدة الجذريّة إلى الطابور.



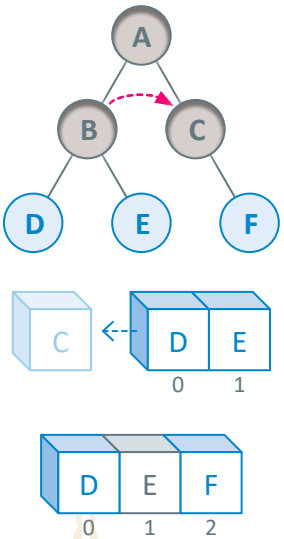
2 احذف العُقدة الجذريّة من الطابور لمعالجتها، ثم أضف فروع هذه العُقدة إلى الطابور (العُقدتين B وC).



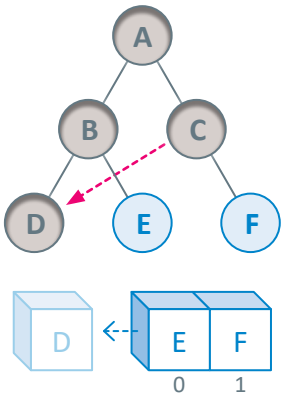
3 احذف العُقدة من مقدمة الطابور (العُقدة B) لمعالجتها، ثم أضف فروع هذه العُقدة إلى الطابور (العُقدتين D وE).



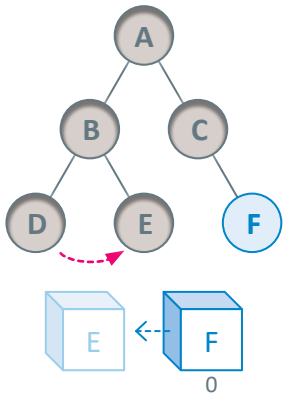
4 احذف العُقدة C وعالجها، ثم أضف فرعها إليها.



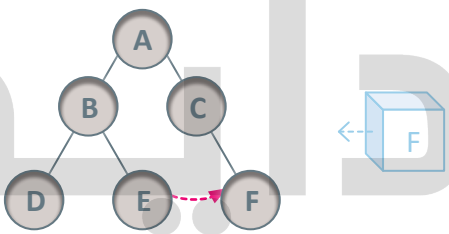
5 احذف العُقدة D لمعالجتها. (ليس لديها فروع).



6 احذف العُقدة E لمعالجتها. (ليس لديها فروع).



7 احذف العُقدة F لمعالجتها، وبذلك أصبح الطابور الآن فارغًا وانتهت عملية البحث.



العُقد التي فُحصَت باستخدام خوارزمية البحث بأولوية الاتساع (BFS) هي: A، B، C، D، E، F.

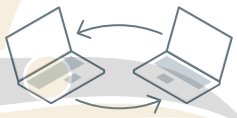
لاحظ كيف يُمكنك تطبيق خوارزمية **البحث بأولوية الاتساع** (BFS) بلغة **البايثون** (Python) في المثال التالي:

```
graph = {
    "A" : ["B","C"],
    "B" : ["D","E"],
    "C" : ["F"],
    "D" : [],
    "E" : [],
    "F" : []
}

visitedBFS = [] # List to keep track of visited nodes
queue = []      # Initialize a queue

 # bfs function
def bfs(visited, graph, node):
    visited.append(node)
```

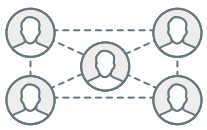
```
  queue.append(node)

  while queue:
    n = queue.pop(0)
    print (n, end = " ")

    for neighbor in graph[n]:
      if neighbor not in visited:
        visited.append(neighbor)
        queue.append(neighbor)

# main program
bfs(visitedBFS, graph, "A")
```

```
A B C D E F
```

التطبيقات العملية لخوارزمية البحث بأولوية الاتساع
Practical Applications of the BFS Algorithm



تُستخدَم في **شبكات النّظير للنّظير** (Peer-to-Peer Networks) للعثور على كل العُقد المجاورة من أجل تأسيس الاتصال.



تُستخدَم في **وسائل التواصل الاجتماعي** (Social Media) لربط عُقد المُستخدِمين المُرتبطين، مثل أولئك الذين لهم الاهتمامات نفسها أو الموقع نفسه.

تُستخدَم في **نُظم الملاحة باستخدام مُحدِّد المواقع العالمي** (GPS Navigation Systems) للبحث عن الأماكن المتجاورة حتى تُحدِّد الاتجاهات التي يتبعها المُستخدِم.

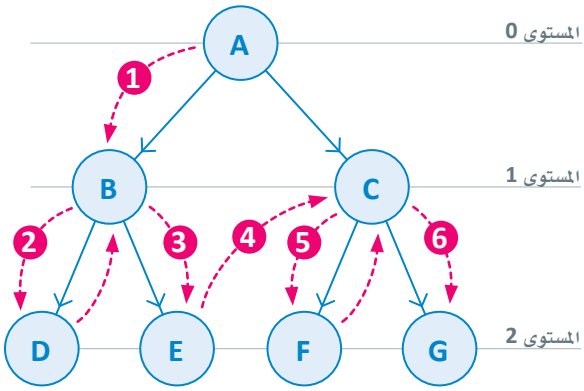
تُستخدَم للحصول على **البث الشبكي** (Network Broadcasting) لبعض الحُزم.

معلومة

يُمكن تطوير خوارزمية البحث بأولوية الاتساع (BFS) بتحديد نقطة البداية (الحالة الأوليّة) ونقطة الهدف (الحالة المُستهدَفة) لإيجاد المسار بينهما.

خوارزمية البحث بأولوية العمق
Depth-First Search (DFS) Algorithm

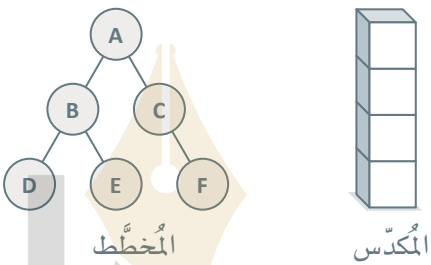
في **البحث بأولوية العمق** (DFS)، ستقوم باتباع الحواف، وتتعمق أكثر وأكثر في المُخطَّط. يَستخدِم البحث بأولوية العمق إجراء استدعاء تكراري للتنقل عبر العُقد. عند الوصول إلى عُقدة لا تحتوي على حواف لأي عُقدة جديدة، ستعود إلى العُقدة السابقة وتستمر العملية. تَستخدِم خوارزمية البحث بأولوية العمق هيكل بيانات المُكدّس لتتبع مسار الاستكشاف. بمجرد استكشاف عُقدة، ستُضاف إلى المُكدّس. عندما ترغب في العودة، ستحذف العُقدة من المُكدّس كما هو موضح في الشكل 2.7.



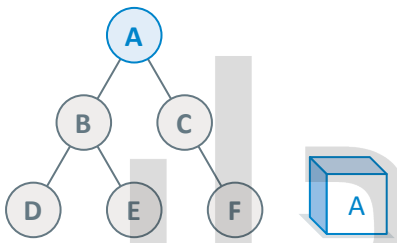
شكل 2.7: خوارزمية البحث بأولوية العمق (DFS)

المثال التالي يوضح طريقة عمل خوارزمية **البحث بأولوية العمق** (DFS)، باستخدام المُخطَّط التالي، تَتَبّع ترتيب استكشاف العُقد (Traversal) بحسب خوارزمية البحث بأولوية العمق.

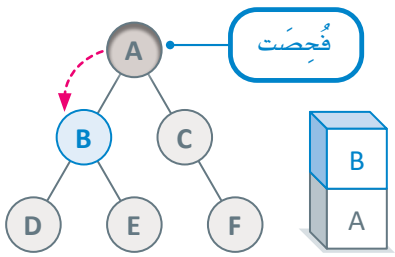
ملاحظة: استخدِم هيكل البيانات المُناسب.



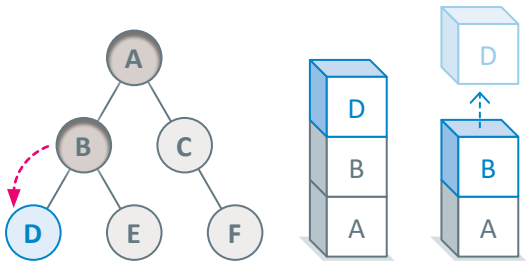
❶ عالِج الجذر A ثم أضفه إلى المُكدّس.



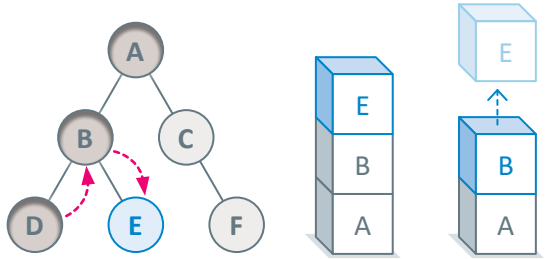
❷ عالِج العُقدة B ثم أضفها إلى المُكدّس.



❸ عالِج العُقدة D ثم أضفها إلى المُكدّس. ستُحذَف العُقدة التي فُحِصَت وليس لها فروع من المُكدّس. (احذف العُقدة D).



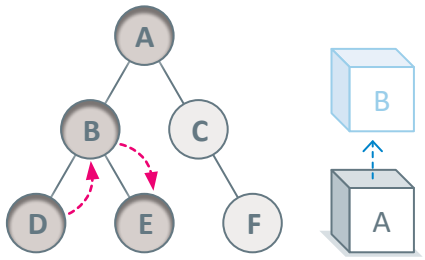
❹ عالِج العُقدة E ثم أضفها إلى المُكدّس. ستُحذَف العُقدة التي فُحِصَت وليس لها فروع من المُكدّس. (احذف العُقدة E).



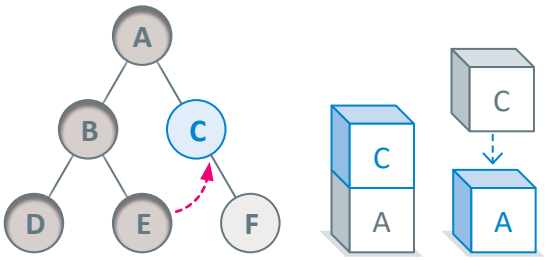
لمحة تاريخية

طُورّت النسخة الأولى من خوارزمية البحث بأولوية العمق (DFS) في القرن التاسع عشر بواسطة عالم رياضيات فرنسي كاستراتيجية لحل المتاهات.

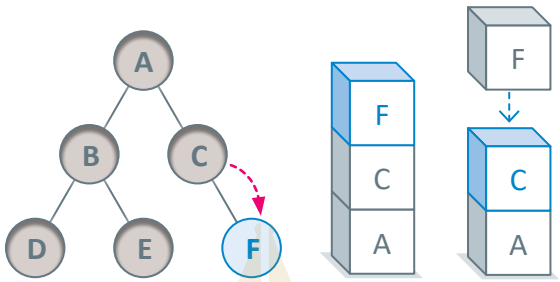
❺ احذف العُقدة B.



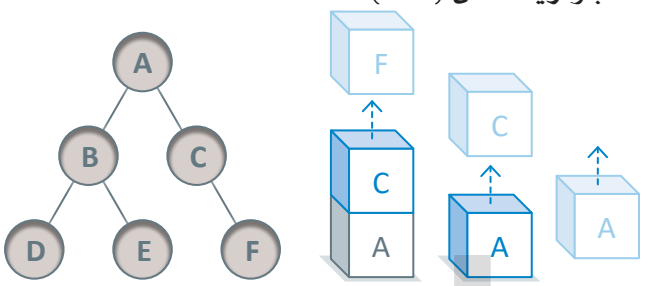
❻ عالِج العُقدة C ثم أضفها إلى المُكدّس.



❼ عالِج العُقدة F ثم أضفها إلى المُكدّس.



❽ المُكدّس خالي وبالتالي ستتوقف خوارزمية **البحث بأولوية العمق** (DFS).



العُقد التي فُحصَت باستخدام خوارزمية البحث بأولوية العمق (DFS) هي: A، B، D، E، C، F.

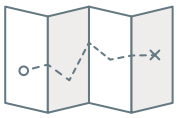
والآن ستتعلّم طريقة تنفيذ خوارزمية **البحث بأولوية العمق** (DFS) في لغة البايثون.


```
graph = {
    "A" : ["B","C"],
    "B" : ["D","E"],
    "C" : ["F"],
    "D" : [],
    "E" : [],
    "F" : []
}

visitedDFS = [] # list to keep track of visited nodes

# dfs function
def dfs(visited, graph, node):
    if node not in visited:
        print(node, end = " ")
        visited.append(node)
        for neighbor in graph[node]:
            dfs(visited, graph, neighbor)


# main program
dfs(visitedDFS, graph, "A")
```

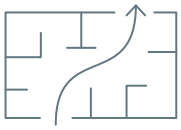
يُستخدم المُكدّس بصورة غير مباشرة عبر مُكدّس أثناء التشغيل (Runtime Stack) لتتبُّع الاستدعاءات التكرارية.

```
A B D E C F
```

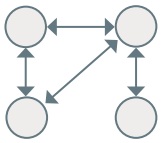
التطبيقات العملية لخوارزمية البحث بأولوية العمق
Practical Applications of the DFS Algorithm



تُستخدَم خوارزمية البحث بأولوية العمق في **إيجاد المسارات** (Path Finding) لاستكشاف المسارات المختلفة في العمق للخرائط والطرقات والبحث عن المسار الأفضل.



تُستخدَم خوارزمية البحث بأولوية العمق في **حل المتاهات** (Solve Mazes) من خلال اجتياز كل الطُرُق الممكنة.



يُمكِن تحديد **الدورات** (Cycles) في المُخطَّط باستخدام خوارزمية البحث بأولوية العمق من خلال وجود **حافة خلفية** (Back Edge)، تمُر من خلال العُقدة نفسها مرتين.

جدول 2.4: مقارنة بين خوارزمية البحث بأولوية الاتساع (BFS) و خوارزمية البحث بأولوية العمق (DFS)

معايير المقارنة	خوارزمية البحث بأولوية الاتساع (BFS)	خوارزمية البحث بأولوية العمق (DFS)
طريقة التنفيذ	التنقّل حسب مستوى الشجرة.	التنقّل حسب عُمق الشجرة.
هيكل البيانات	تَستخدِم هيكل بيانات الطابور لتتبُّع الموقع التالي لفحصه.	تَستخدِم هيكل بيانات المُكدّس لتتبُّع الموقع التالي لفحصه.
الاستخدام	يُفضَّل استخدامها عندما يكون هيكل المُخطَّط واسعًا وقصيرًا.	يُفضَّل استخدامها عندما يكون هيكل المُخطَّط ضيقًا وطويلًا.
طريقة البحث	تبحث عن مسار الوجهة باستخدام أقل عدد من الحواف.	يتجة البحث إلى قاع الشجرة الفرعية، ثم يتراجع.
العُقد التي تُفحص في البداية	فحص عُقد الأشقاء قبل الفروع.	فحص عُقد الفروع قبل الأشقاء.

تمرينات

1

حدِّد الجملة الصحيحة والجملة الخاطئة فيما يلي:	صحيحة	خاطئة
1. تُنفَّذ خوارزمية البحث بأولوية الاتساع (BFS) وخوارزمية البحث بأولوية العمق (DFS) باستخدام الاستدعاء الذاتي.	✓	
2. لا يمكن استخدام خوارزمية البحث بأولوية الاتساع (BFS) وخوارزمية البحث بأولوية العمق (DFS) في هيكل بيانات الشجرة.		✓
3. تُنفَّذ خوارزمية البحث بأولوية الاتساع (BFS) بمساعدة هيكل بيانات القائمة المترابطة.	✓	
4. يمكن تنفيذ خوارزمية البحث بأولوية العمق (DFS) بمساعدة هيكل بيانات المُكدّس.	✓	
5. لا يمكن استخدام خوارزمية البحث بأولوية الاتساع (BFS) في البث الشبكي.		✓

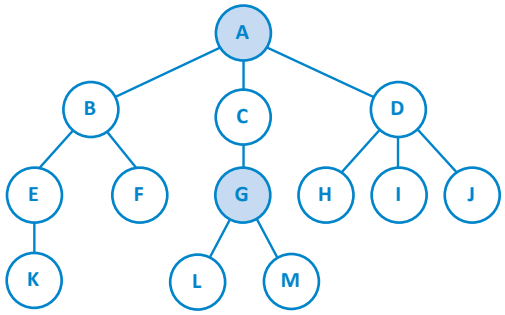
2 اشرح كيف تعمل خوارزمية البحث بأولوية الاتساع (BFS) وخوارزمية البحث بأولوية العمق (DFS).

يمكن للطلبة الرجوع لصفحة 84،79 من الكتاب لحل هذا السؤال

3 قارن بين خوارزمية البحث بأولوية الاتساع (BFS) وخوارزمية البحث بأولوية العمق (DFS).

يمكن للطلبة الرجوع لصفحة 85 من الكتاب لحل هذا السؤال

4 في المُخطَّط على اليسار، انتقِل من عُقدة البداية A إلى عُقدة الهدف G. طبِّق خوارزمية البحث بأولوية الاتساع (BFS) وخوارزمية البحث بأولوية العمق (DFS) باستخدام هيكل البيانات المناسب (المُكدّس أو الطابور)، مع الإشارة إلى العُقد التي فُحِصَت.



BFS :

A-> B-> C-> D-> E-> F-> G

بالنسبة إلى خوارزمية البحث بأولوية الاتساع BFS سنستخدم قائمة طابوراً :

push A

pop A (checked)

push B

push C

push D

pop B (checked)

push E

push F

pop C (checked)

push G

pop D (checked)

push H

push I

push J

pop E (checked)

push K

pop F (checked)

pop G (checked)

DFS :

A-> B->E->K->F-> C->G

بالنسبة إلى خوارزمية البحث بأولوية العمق DFS

push A (checked)

push B (checked)

push E (checked)

push K (checked)

pop K

pop E

push F (checked)

pop F

pop B

push C (checked)

puch G (checked)

5 اكتب دالة بلغة البايثون تَستخدِم خوارزمية البحث بأولوية الاتساع (BFS) في مُخطَّط للتحقق مما إذا كان هناك مسارٌ بين عُقدتين مُعطاتين.

6 اكتب دالة بلغة البايثون تَستخدِم خوارزمية البحث بأولوية العمق (DFS) لإيجاد المسار الأقصر في مخطط غير موزون.

رابط الدرس الرقمي

www.ien.edu.sa

الدرس الثالث

اتخاذ القرار القائم على القواعد

الأنظمة القائمة على القواعد Rule-Based Systems

تُركِّز أنظمة الذكاء الاصطناعي القائمة على القواعد على استخدام مجموعة من القواعد المُحدَّدة مُسبقًا لاتخاذ القرارات وحل المشكلات. **الأنظمة الخبيرة** (Expert Systems) هي المثال الأكثر شهرة للذكاء الاصطناعي القائم على القواعد، وهي إحدى صور الذكاء الاصطناعي الأولى التي طُوِّرت وانتشرت في فترة الثمانينيات والتسعينيات من القرن الماضي. وغالبًا ما كانت تُستخدَم لأتمتة المهام التي تتطلب عادةً خبرات بشرية مثل: تشخيص الحالات الطبية أو تحديد المشكلات التقنية وإصلاحها. واليوم لم تَعُد الأنظمة القائمة على القواعد التقنية هي الأحدث، حيث تفوّقت عليها منهجيات الذكاء الاصطناعي الحديثة. ومع ذلك، لا تزال الأنظمة الخبيرة شائعة الاستخدام في العديد من المجالات نظرًا لقدرتها على الجمع بين الأداء المعقول وعملية اتخاذ القرار البديهية والقابلة للتفسير.

قاعدة المعرِفة Knowledge Base

أحد المكونات الرئيسة لأنظمة الذكاء الاصطناعي القائمة على القواعد هي قاعدة المعرِفة، وهي مجموعة من الحقائق والقواعد التي يَستخدِمها النظام لاتخاذ القرارات. تُدخَل هذه الحقائق والقواعد في النظام بواسطة الخبراء البشريين المسؤولين عن تحديد المعلومات الأكثر أهمية وتحديد القواعد التي يتَّبعها النظام. لاتخاذ القرار أو حل المُشكلة، يبدأ النظام الخبير بالتحقق من الحقائق والقواعد في قاعدة البيانات و تطبيقها على الموقف الحالي. إن لم يتمكن النظام من العثور على تطابق بين الحقائق والقواعد في قاعدة المعرِفة، فقد يطلب من المُستخدِم معلومات إضافية أو إحالة المشكلة إلى خبير بشري لمزيد من المساعدة، وإليكَ بعض مزايا وعيوب الأنظمة القائمة على القواعد موضحة في جدول 2.5:

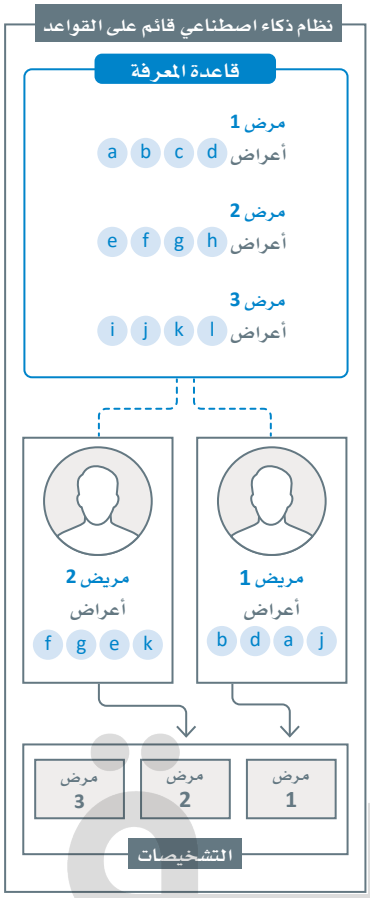
الأنظمة الخبيرة (Expert systems):

النظام الخبير هو أحد أنواع الذكاء الاصطناعي الذي يُحاكي قدرة اتخاذ القرار لدى الخبير البشري. يَستخدِم النظام قاعدة المعرِفة المُكوَّنة من قواعد وحقائق ومحركات الاستدلال لتقديم المشورة أو حل المشكلات في مجال معرفي مُحدَّد.

جدول 2.5: المزايا والعيوب الرئيسة للأنظمة القائمة على القواعد

المزايا	العيوب
• يُمكنها اتخاذ القرارات وحل المشكلات بسرعة وبدقة أفضل من البشر، خاصةً عندما يتعلق الأمر بالمهام التي تتطلب قدرًا كبيرًا من المعرِفة أو البيانات. • تَعمل هذه الأنظمة باستمرار، دون تحيُّز أو أخطاء قد تؤثر في بعض الأحيان على اتخاذ القرار البشري.	• تَعمل هذه الأنظمة بكفاءة طالما كانت مُدخَلات المعرِفة والقواعد جيدة، وقد لا تستطيع التعامل مع المواقف التي تقع خارج نطاق خبراتها. • لا يُمكنها التعلُّم أو التكيُّف بالطريقة نفسها مثل البشر، وهذا يجعلها أقل قابلية للتطبيق على الأحداث المُتغيِّرة حيث تتغير مُدخَلات البيانات والمنطق كثيرًا بمرور الوقت.

في هذا الدرس ستتعلّم المزيد حول الأنظمة القائمة على القواعد في سياق أحد تطبيقاتها الرئيسة، وهو: التشخيص الطبي. سيعرض النظام تشخيصًا طبيًا وفقًا للأعراض التي تظهر على المريض، كما هو مُوضّح في الشكل 2.8. بدءًا بنظام تشخيص بسيط مُستنِد إلى القواعد، وستكتشف بعض الأنظمة الأكثر ذكاءً وكيف يُحقّق كل تكرار نتائج أفضل.



شكل 2.8: التشخيص الطبي بواسطة نظام الذكاء الاصطناعي القائم على القواعد

الإصدار 1

في الإصدار الأول ستبني نظامًا بسيطًا قائمًا على القواعد يمكنه تشخيص ثلاثة أمراض مُحتملة: KidneyStones (حصى الكُلى)، وAppendicitis (التهاب الزائدة الدودية)، وFood Poisoning (التسمُّم الغذائي). ستكون المُدخَلات إلى النظام هي قاعدة معرفة بسيطة تربط كل مرض بقائمة من الأعراض المُحتملة. يتوفّر ذلك في ملف بتنسيق JSON (جيسون) يُمكنك تحميله وعرضه كما هو مُوضَّح بالأسفل.

```
import json  # a library used to save and load JSON files

# the file with the symptom mapping
symptom_mapping_file='symptom_mapping_v1.json'

# open the mapping JSON file and load it into a dictionary
with open(symptom_mapping_file) as f:
    mapping=json.load(f)

# print the JSON file
print(json.dumps(mapping, indent=2))
```



```
{
  "diseases": {
    "food poisoning": [
      "vomiting",
      "abdominal pain",
      "diarrhea",
      "fever"
    ],
    "kidney stones": [
      "lower back pain",
      "vomiting",
      "fever"
    ],
    "appendicitis": [
      "abdominal pain",
      "vomiting",
      "fever"
    ]
  }
}
```

سيتَّبع الإصدار الأول القائم على القواعد قاعدة بسيطة ألا وهي: إذا كان لدى المريض على الأقل ثلاثًا من جميع الأعراض المحتملة للمرض، فيجب إضافة المرض كتشخيص مُحتَمل. يمكنك العثور أدناه على دالة Python (البايثون) التي تَستخدِم هذه القاعدة لإجراء التشخيص، بالاستناد إلى قاعدة المعرِفة المذكورة أعلاه وأعراض المرض الظاهرة على المريض.

```
def diagnose_v1(patient_symptoms:list):

    diagnosis=[] # the list of possible diseases

    if "vomiting" in patient_symptoms:

        if "abdominal pain" in patient_symptoms:

            if "diarrhea" in patient_symptoms:

                    # 1:vomiting, 2:abdominal pain, 3:diarrhea
                    diagnosis.append('food poisoning')

            elif 'fever' in patient_symptoms:

                    # 1:vomiting, 2:abdominal pain, 3:fever
                    diagnosis.append('food poisoning')
                    diagnosis.append('appendicitis')

        elif "lower back pain" in patient_symptoms and 'fever' in patient_symptoms:
            # 1:vomiting, 2:lower back pain, 3:fever
            diagnosis.append('kidney stones')

    elif "abdominal pain" in  patient_symptoms and\
         "diarrhea" in patient_symptoms and\
         "fever" in patient_symptoms:\
        # 1:abdominal pain, 2:diarrhea, 3:fever
        diagnosis.append('food poisoning')

    return diagnosis
```


في هذه الحالة، تكون قاعدة المعرِفة **محددةً بتعليمات برمجية ثابتة** (Hard-Coded) داخل الدالة في شكل عبارات IF. تَستخدِم هذه العبارات الأعراض الشائعة بينَ الأمراض الثلاثة للتوصل تدريجيًا إلى التشخيص في أسرع وقت ممكن. على سبيل المثال، عَرض Vomiting (القيء) مشترك بين جميع الأمراض. لذلك، إذا كانت عبارة IF الأولى صحيحة فقد تم بالفعل حساب أحد الأعراض الثلاثة المطلوبة لجميع الأمراض. بعد ذلك، سوف تبدأ في البحث عن Abdominal Pain (ألم البطن) المرتبط بمرضين وتستمر بالطريقة نفسها حتى يتم النظر في جميع مجموعات الأعراض الممكنة.

يُمكنك بعد ذلك اختبار هذه الدالة على ثلاثة مرضى مختلفين:

```
# Patient 1
my_symptoms=['abdominal pain', 'fever', 'vomiting']
diagnosis=diagnose_v1(my_symptoms)
print('Most likely diagnosis:',diagnosis)

# Patient 2
my_symptoms=['vomiting', 'lower back pain', 'fever' ]
diagnosis=diagnose_v1(my_symptoms)
print('Most likely diagnosis:',diagnosis)

# Patient 3
my_symptoms=['fever', 'cough', 'vomiting']
diagnosis=diagnose_v1(my_symptoms)
print('Most likely diagnosis:',diagnosis)
```

```
Most likely diagnosis: ['food poisoning', 'appendicitis']
Most likely diagnosis: ['kidney stones']
Most likely diagnosis: []
```

شكل 2.9: تمثيل الإصدار الأول

يتضمن التشخيص الطبي للمريض الأول التسمُّم الغذائي والتهاب الزائدة الدودية لأن الأعراض الثلاثة التي تظهر على المريض ترتبط بكلا المرضين. يُشخَّص المريض الثاني بحصى الكُلى، فهو المرض الوحيد الذي تجتمع فيه الأعراض الثلاثة. في النهاية، لا يُمكن تشخيص الحالة الطبية للمريض الثالث؛ لأن الأعراض الثلاثة التي ظهرت على المريض لا تجتمع في أي من الأمراض الثلاثة.

يتميز الإصدار الأول القائم على القواعد بالبديهية والقابلية للتفسير، كما يتضمن استخدام قاعدة المعرِفة والقواعد في التشخيص الطبي دون تَحيُّز أو انحراف عن الخط المعياري. ومع ذلك، يشوب هذا الإصدار العديد من العيوب: أولًا، أن قاعدة ثلاثة أعراض على الأقل هي تمثيل مُبسَّط للغاية لكيفية التشخيص الطبي على يد الخبير البشري. ثانيًا، أن قاعدة المعرِفة داخل الدالة تكون محددةً بتعليمات برمجية ثابتة، وعلى الرغم من أنه يسهُل إنشاء عبارات شَرطيَّة بسيطة لقواعد المعرِفة الصغيرة، إلا أن المهمة تصبحُ أكثر تعقيدًا وتستغرق وقتًا طويلًا عند تشخيص الحالات التي تعاني من العديد من الأمراض والأعراض المرضية.

الإصدار 2

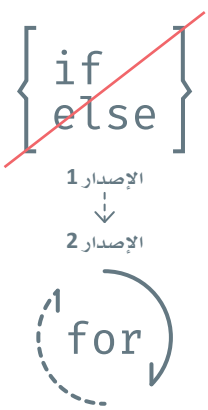
في الإصدار الثاني، ستُعزِّز مرونة وقابلية تطبيق النظام القائم على القواعد بتمكينه من قراءة قاعدة المعرِفة المُتغيِّرة مباشرةً من ملف JSON (جسون). سيؤدي هذا إلى الحد من عملية الهندسة اليدوية لعبارات IF الشَرطيَّة حسب الأعراض ضمن الدالة. وهذا يُعدُّ تحسُّنًا كبيرًا يجعل النظام قابلًا للتطبيق على قواعد المعرِفة الأكبر حجمًا مع تزايد عدد الأمراض والأعراض. وفي الأسفل، مثال يوضّح قاعدة المعرِفة.

```
symptom_mapping_file='symptom_mapping_v2.json'

with open(symptom_mapping_file) as f:
    mapping=json.load(f)

print(json.dumps(mapping, indent=2))
```

```
{
  "diseases": {
    "covid19": [
      "fever",
      "headache",
      "tiredness",
      "sore throat",
      "cough"
    ],
    "common cold": [
      "stuffy nose",
      "runny nose",
      "sneezing",
      "sore throat",
      "cough"
    ],
    "flu": [
      "fever",
      "headache",
      "tiredness",
      "stuffy nose",
      "sneezing",
      "sore throat",
      "cough",
      "runny nose"
    ],
    "allergies": [
      "headache",
      "tiredness",
      "stuffy nose",
      "sneezing",
      "cough",
      "runny nose"
    ]
  }
}
```


قاعدة المعرِفة الجديدة هذه أكبر قليلًا من سابقتها. ومع ذلك، يتَّضح أن محاولة إنشاء عبارات IF الشَرطيَّة في هذه الحالة ستكون أصعب بكثير. على سبيل المثال، تضمنت قاعدة المعرِفة السابقة ربط أحد الأمراض بأربعة أعراض، ومرضين بثلاثة أعراض. وعند تطبيق قاعدة ثلاثة أعراض على الأقل المُطبَّقة في الإصدار الأول، تحصل على 6 مجموعات ثلاثية من الأعراض المحتملة التي تؤخَذ في الاعتبار. في قاعدة المعرِفة الجديدة بالأعلى، تكون للأمراض الأربعة 5 و5 و8 و6 أعراض، على التوالي. وبهذا، تحصل على 96 مجموعة ثلاثية من الأعراض المحتملة. وفي حال التعامل مع مئات أو حتى آلاف الأمراض، ستجدُ أنّه من المستحيل إنشاء نظام مثل الموجود في الإصدار الأول.

وكذلك، لا يوجد سبب طبي وجيه لقِصَر التشخيص الطبي على مجموعات ثلاثية من الأعراض. ولذلك، ستجعل **منطق التشخيص** (Diagnosis Logic) أكثر تنوعًا بحساب عدد الأعراض المُطابقة لكل مرض، والسماح للمُستخدِم بتحديد عدد الأعراض المُطابقة التي يجب توافرها في المرض لتضمينه في التشخيص.



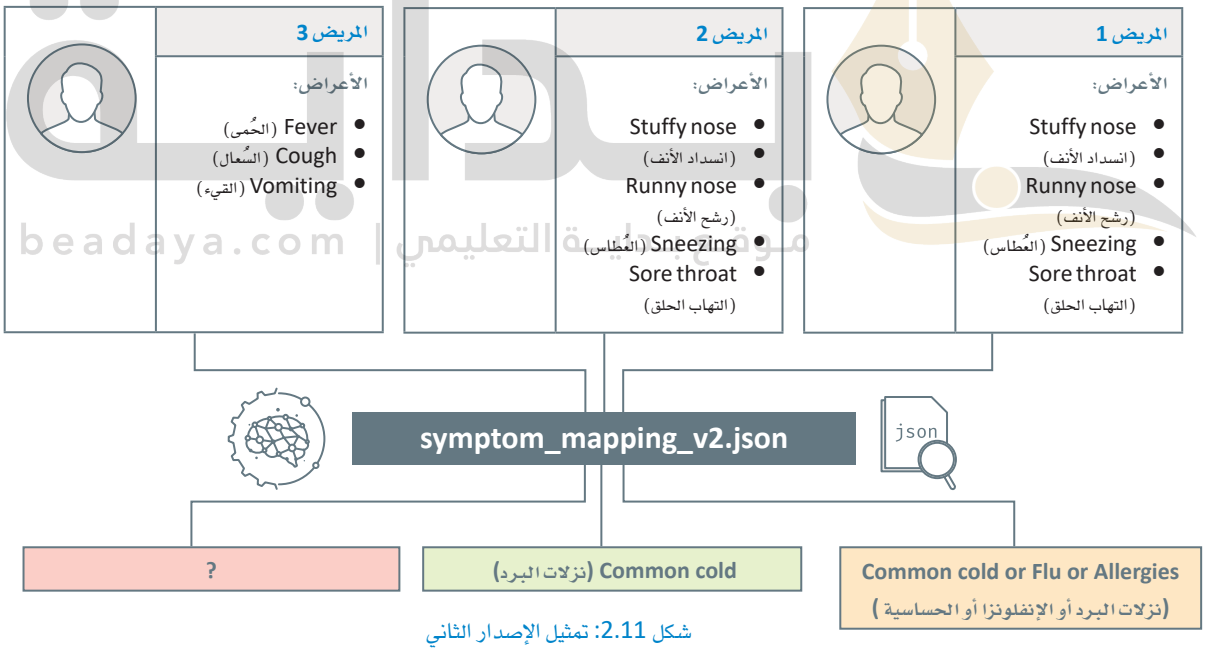
شكل 2.10: الإصدار الثاني لا يحتوي على عبارات IF الشَرطيَّة المحددةً بتعليماتٍ برمجيةٍ ثابتة.

```
def diagnose_v2(patient_symptoms:list,
                symptom_mapping_file:str,
                matching_symptoms_lower_bound:int):

    diagnosis=[]

    with open(symptom_mapping_file) as f:
        mapping=json.load(f)

    # access the disease information
    disease_info=mapping['diseases']

    # for every disease
    for disease in disease_info:

        counter=0

        disease_symptoms=disease_info[disease]

        # for each patient symptom
        for symptom in patient_symptoms:

            # if this symptom is included in the known symptoms for the disease
            if symptom in disease_symptoms:
                counter+=1

        if counter>=matching_symptoms_lower_bound:
            diagnosis.append(disease)

    return diagnosis
```


لا يحتوي هذا الإصدار على عبارات IF الشَرطيَّة المحددة بتعليماتٍ برمجيةٍ ثابتة. بعد تحميل مُخطَّط الأعراض من ملف JSON (جسون)، يبدأ الإصدار في أخذ كلّ مرض محتمل في الاعتبار باستخدام حلقة التكرار الأولى FOR. تتحقق الحلقة من كل عَرَض على حدة بمقارنته بالأعراض المعروفة للمرض وزيادة **العدَّاد** (Counter) في كل مرة يجد فيها النظام تطابقًا.

```python
# Patient 1
my_symptoms=["stuffy nose", "runny nose", "sneezing", "sore throat"]
diagnosis=diagnose_v2(my_symptoms,'symptom_mapping_v2.json' , 3)
print('Most likely diagnosis:',diagnosis)

# Patient 2
my_symptoms=["stuffy nose", "runny nose", "sneezing", "sore throat"]
diagnosis=diagnose_v2(my_symptoms, 'symptom_mapping_v2.json' , 4)
print('Most likely diagnosis:',diagnosis)

# Patient 3
my_symptoms=['fever', 'cough', 'vomiting']
diagnosis=diagnose_v2(my_symptoms, 'symptom_mapping_v2.json' , 3)
print('Most likely diagnosis:',diagnosis)
```

```
Most likely diagnosis: ['common cold', 'flu', 'allergies']
Most likely diagnosis: ['common cold']
Most likely diagnosis: []
```



شكل 2.11: تمثيل الإصدار الثاني

لاحظ أن الإصدار الثاني هو نسخة مُعمَّمة من الإصدار الأول. ومع ذلك، يُعدُّ هذا الإصدار أكثر قابلية للتطبيق على نطاق واسع، ويمكن استخدامه كما هو مع أي قاعدة معرِفة أخرى بالتنسيق نفسه، حتى لو كانت تشمل الآلاف من الأمراض مع عدد ضخم من الأعراض. كما يَسمح للمُستخدِم بزيادة أو تقليل عدد القيود على التشخيص بضبط المُتغيِّر matching_symptoms_lower_bound. يمكن ملاحظة ذلك في حالة المريض 1 والمريض 2: فعلى الرغم من أنهما يعانيان من الأعراض نفسها، إلا أنه عند ضبط هذا المُتغيِّر، ستحصل على تشخيص مختلف تمامًا.

على الرغم من هذه التحسينات،إلا إنّ بعض العيوب لا تزال موجودة في هذا الإصدار، ولا يُعدُّ تمثيلًا دقيقًا للتشخيص الطبي الحقيقي.

الإصدار 3

في الإصدار الثالث، ستزيد من ذكاء النظام القائم على القواعد بمنحه إمكانية الوصول إلى نوع مُفصَّل من قاعدة المعرِفة. هذا النوع الجديد يأخذ بعين الاعتبار الحقيقة الطبية التي تقول: إنّ بعض الأعراض تكون أكثر شيوعًا من أخرى للمرض نفسه.

```
symptom_mapping_file='symptom_mapping_v3.json'

with open(symptom_mapping_file) as f:
    mapping=json.load(f)

print(json.dumps(mapping, indent=2))
```

```
{
  "diseases": {
    "covid19": {
      "very common": [
        "fever",
        "tiredness",
        "cough"
      ],
      "less common": [
        "headache",
        "sore throat"
      ]
    },
    "common cold": {
      "very common": [
        "stuffy nose",
        "runny nose",
        "sneezing",
        "sore throat"
      ],
      "less common": [
        "cough"
      ]
    },
   "flu": {
      "very common": [
        "fever",
        "headache",
        "tiredness",
        "sore throat",
        "cough"
      ],
      "less common": [
        "stuffy nose",
        "sneezing",
        "runny nose"
      ]
    },
    "allergies": {
      "very common": [
        "stuffy nose",
        "sneezing",
        "runny nose"
      ],
      "less common": [
        "headache",
        "tiredness",
        "cough"
      ]
    }
  }
}
```

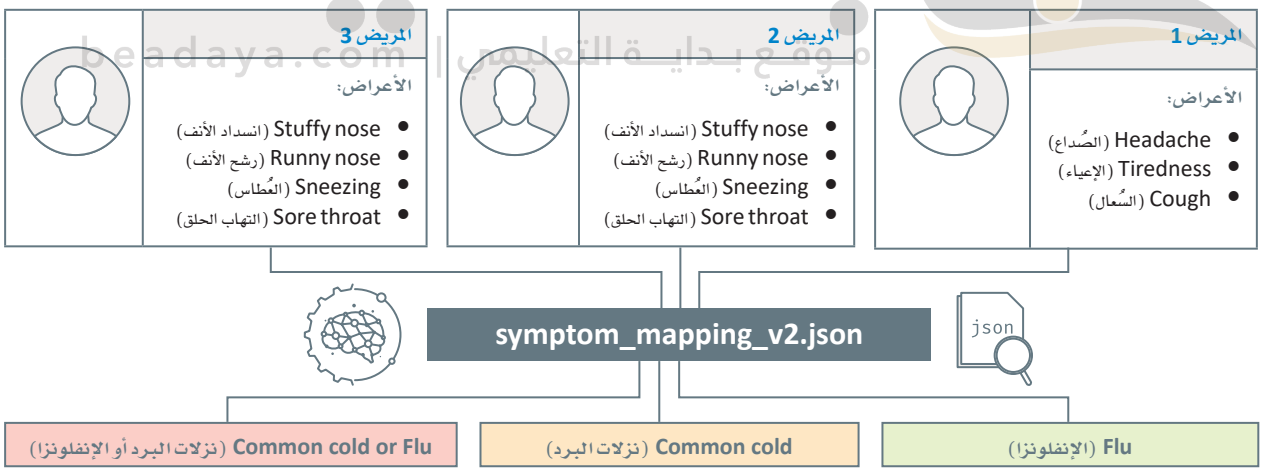
لن يُنظر إلى المنطق الذي يقتصر على عدد الأعراض، وسيُستبدَل بدالة تسجيل النقاط التي تعطي أوزانًا مُخصَّصة للأعراض الأكثر والأقل شيوعًا. ستَتوفر للمستخدِم كذلك المرونة لتحديد الأوزان التي يراها مناسبة. سيتمّ تضمين المرض أو الأمراض ذات المجموع الموزون الأعلى في التشخيص.

```
from collections import defaultdict

def diagnose_v3(patient_symptoms:list,
                symptom_mapping_file:str,
                very_common_weight:float=1,
                less_common_weight:float=0.5
               ):

    with open(symptom_mapping_file) as f:
        mapping=json.load(f)

    disease_info=mapping['diseases']

    # holds a symptom-based score for  each potential disease
    disease_scores=defaultdict(int)

    for disease in disease_info:

        # get the very common symptoms of the disease
        very_common_symptoms=disease_info[disease]['very common']

        # get the less common symptoms for this disease
        less_common_symptoms=disease_info[disease]['less common']

        for symptom in patient_symptoms:

            if symptom in very_common_symptoms:
                disease_scores[disease]+=very_common_weight

            elif symptom in less_common_symptoms:
                disease_scores[disease]+=less_common_weight

    # find the max score all candidate diseases
    max_score=max(disease_scores.values())

    if max_score==0:
        return []

    else:
        # get all diseases that have the max score
        diagnosis=[disease for disease in disease_scores if disease_scores
[disease]==max_score]

        return diagnosis, max_score
```

لكل مرض محتمل في قاعدة المعرفة، تُحدِّد هذه الدالة الجديدة الأعراض الأكثر والأقل ظهورًا على المريض، ثم تزيد من درجة المرض وفقًا للأوزان المُقابِلة، وفي الأخير تُدرَج الأمراض ذات الدرجة الأعلى في التشخيص. يُمكنك الآن اختبار تنفيذ الدالة مع بعض الأمثلة:

```
# Patient 1
my_symptoms=["headache", "tiredness", "cough"]
diagnosis=diagnose_v3(my_symptoms, 'symptom_mapping_v3.json')
print('Most likely diagnosis:',diagnosis)

# Patient 2
my_symptoms=["stuffy nose", "runny nose", "sneezing", "sore throat"]
diagnosis=diagnose_v3(my_symptoms, 'symptom_mapping_v3.json')
print('Most likely diagnosis:',diagnosis)

# Patient 3
my_symptoms=["stuffy nose", "runny nose", "sneezing", "sore throat"]
diagnosis=diagnose_v3(my_symptoms, 'symptom_mapping_v3.json', 1, 1)
print('Most likely diagnosis:',diagnosis)
```

```
Most likely diagnosis: (['flu'], 3)
Most likely diagnosis: (['common cold'], 4)
Most likely diagnosis: (['common cold', 'flu'], 4)
```



شكل 2.12: تمثيل الإصدار الثالث

قد تلاحظ أنه على الرغم من أن الأعراض الثلاثة على المريض 1: Headache (الصداع)، وTiredness (الإعياء)، وCough (السعال) تظهر عند الإصابة بكل من Flu (الإنفلونزا)، و Covid19 (كوفيد- 19). والحساسية، إلّا أنّ الظّاهر في نتائج التّشخيص هي الإنفلونزا فقط. هذا لأن جميع الأعراض الثلاثة شائعة جدًا في قاعدة المعرفة، مما يؤدي إلى درجة قصوى قدرها 3. وبالمثل، في ظل معاناة المريض الثاني والثالث من الأعراض نفسها، تؤدي مُدخَلات الأوزان المختلفة للأعراض الأكثر والأقل شيوعًا إلى تشخيصات مختلفة. وعلى وجه التحديد، يَنتج عن استخدام وزن متساوٍ لنوعين من الأعراض إضافة الإنفلونزا إلى التشخيص.

الإصدار 4

يمكن تحسين النظام القائم على القواعد بزيادة كفاءة قاعدة المعرِفة وتجربة **دوال تسجيل النقاط** (Scoring Functions) المختلفة. وعلى الرغم من أن ذلك سيؤدي إلى تحسين النظام، إلا أنه سيتطلب الكثير من الوقت والجهد اليدوي. ولحسن الحظ، هناك طريقة آلية لبناء نظام مبني على القواعد يكون ذكيًا بما يكفي لتصميم قاعدة معرِفة ودالة تسجيل نقاط خاصة به: باستخدام تعلُّم الآلة. يُطبِّق **تعلُّم الآلة القائم على القواعد** (Rule-Based Machine Learning) خوارزمية تعلّم لتحديد القواعد المُفيدة تلقائيًا، بدلًا من الحاجة إلى الإنسان لتطبيق المعرِفة والخبرات السابقة في المجال لبناء القواعد وتنظيمها يدويًا.

فبدلًا من قاعدة المعرِفة ودالة تسجيل النقاط المُصمَّمتان يدويًا، تَتوقَّع خوارزمية تعلّم الآلة مدخلًا واحدًا فقط وهو مجموعة البيانات التاريخيّة للحالات المَرَضيَّة. فالتعلُّم من البيانات مباشرةً يحوُل دون حدوث المشكلات المرتبطة باكتساب المعرِفة الأساسية والتحقق منها. تتكون كل حالة من بيانات أعراض المريض والتشخيص الطبي الذي يمكن أن يقدمه أي خبير بشري مثل الطبيب. وباستخدام مجموعة بيانات التدريب، تتعلّم الخوارزمية تلقائيًا كيف تتنبأ بالتشخيص المُحتمَل لحالة مريض جديد.

```
import pandas as pd # import pandas to load and process spreadsheet-type data

medical_dataset=pd.read_csv('medical_data.csv') # load a medical dataset.

medical_dataset
```

	fever	cough	tiredness	headache	stuffy nose	runny nose	sneezing	sore throat	diagnosis
0	1	1	1	0	0	0	0	0	covid19
1	0	1	1	1	0	0	0	0	covid19
2	1	1	1	0	0	0	0	0	covid19
3	1	1	1	0	0	0	0	0	covid19
4	1	1	1	0	0	0	0	0	covid19
...	...	...	...	...	...	...	...	...	...
1995	0	1	0	0	1	0	1	1	common cold
1996	0	0	0	1	1	1	1	0	common cold
1997	0	0	1	0	1	0	0	1	common cold
1998	0	0	0	0	1	0	0	1	common cold
1999	0	1	0	0	0	0	1	1	common cold

في المثال أعلاه، تحتوي مجموعة البيانات على 2,000 حالة مرضية، بحيث تتكون كل حالة من 8 أعراض محتملة: Fever (الحُمى)، وCough (السُعال)، وTiredness (الإعياء)، وHeadache (الصُداع)، وStuffy nose (انسداد الأنف)، وRunny nose (رشح الأنف)، وSneezing (العُطاس)، وSore throat (التهاب الحلق). تُرمَّز كل واحدة من هذه الأعراض في عمود ثنائي مُنفصل. العدد الثنائي 1 يشير إلى أن المريض يُعاني من الأعراض، بينما العدد الثنائي 0 يشير إلى أن المريض لا يُعاني من الأعراض.

يحتوي العمود الأخير على تشخيص الخبير البشري، وهناك أربعة تشخيصات محتملة:

Covid19 (كوفيد- 19)، وFlu (الإنفلونزا)، وAllergies (الحساسية)، وCommon cold (نزلات البرد).

يمكنك التحقق من ذلك بسهولة باستخدام المقطع البرمجي التالي بلغة البايثون:

```
set(medical_dataset['diagnosis'])
```

على الرغم من أن هناك العشرات من خوارزميات تعلُّم الآلة المحتملة التي يمكن استخدامها مع مجموعة البيانات هذه، إلا أنك ستستخدم تلك التي تتبع المنهجية المُستنِدة على منطق **شجرة القرار** (Decision Tree)، كما ستَستخدِم DecisionTreeClassifier (مصنّف شجرة القرار) من مكتبة البايثون **سكليرن** (Sklearn) على وجه التحديد.

```
from sklearn.tree import DecisionTreeClassifier

def diagnose_v4(train_dataset:pd.DataFrame):

    # create a DecisionTreeClassifier
    model=DecisionTreeClassifier(random_state=1)

    # drop the diagnosis column to get only the symptoms
    train_patient_symptoms=train_dataset.drop(columns=['diagnosis'])

    # get the diagnosis column, to be used as the classification target
    train_diagnoses=train_dataset['diagnosis']

    # build a decision tree
    model.fit(train_patient_symptoms, train_diagnoses)

    # return the trained model
    return model
```


يُعدُّ تطبيق البايثون في الإصدار الرابع أقصر وأبسط بكثير من التطبيقات السابقة، فهو ببساطة يقرأ الملف التدريبي، ويستخدمه لبناء نموذج شجرة القرار استنادًا إلى العلاقات بين الأعراض والتشخيصات، ومن ثَمَّ ينتج نموذجًا مخصَّصًا. لاختبار هذا الإصدار بشكل صحيح، ابدأ بتقسيم مجموعة البيانات إلى مجموعتين منفصلتين، واحدة للتدريب، وأخرى للاختبار.

```
from sklearn.model_selection import train_test_split

# use the function to split the data, get 30% for testing and 70% for training.
train_data, test_data = train_test_split(medical_dataset, test_size=0.3,
random_state=1)

#print the shapes (rows x columns) of the two datasets
print(train_data.shape)
print(test_data.shape)
```

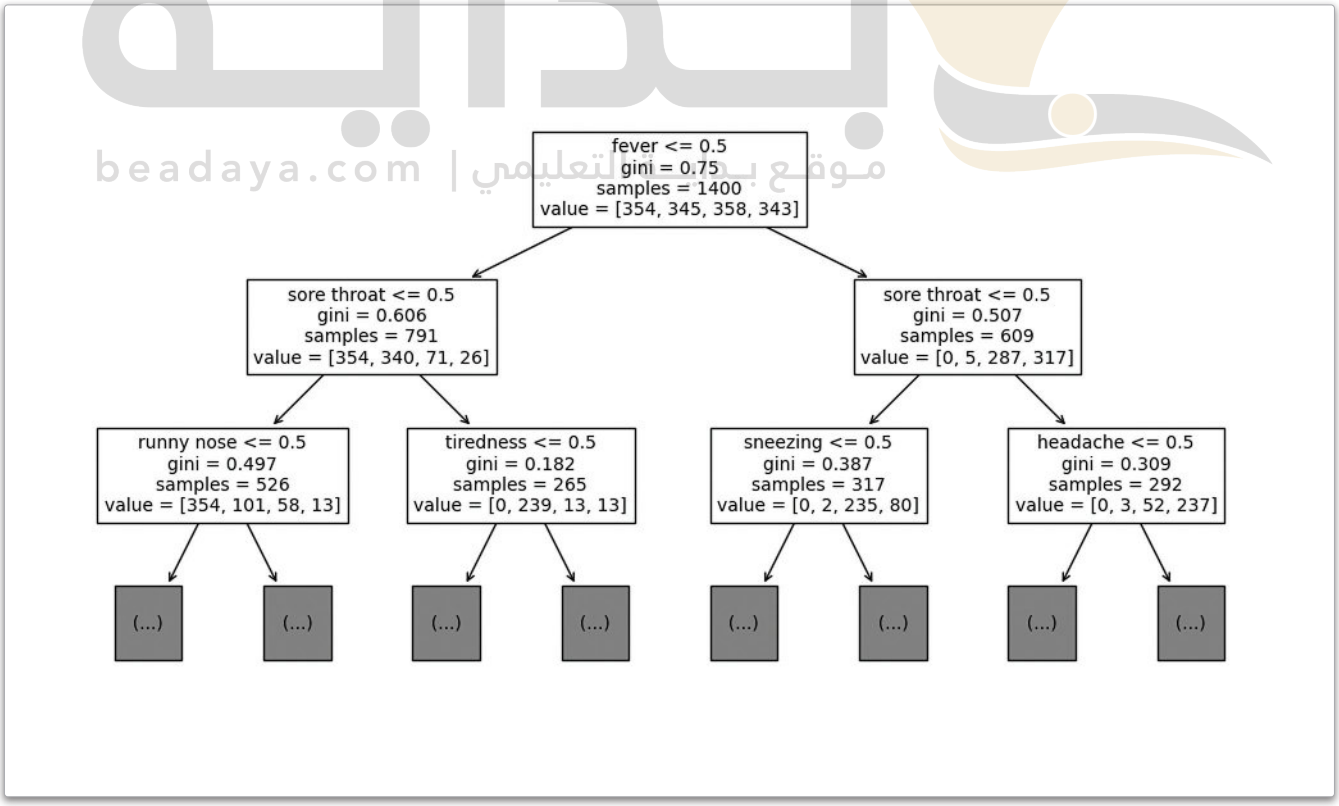
```
(1400, 9)
(600, 9)
```

لديك الآن 1,400 نقطة بيانات ستُستخدَم لتدريب النموذج و600 نقطة ستُستخدَم لاختباره.

ابدأ بتدريب نموذج شجرة القرار وتمثيله:

```
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt

my_tree=diagnose_v4(train_data) # train a model

print(my_tree.classes_) # print the possible target labels (diagnoses)

plt.figure(figsize=(12,6)) # size of the visualization, in inches

# plot the tree
plot_tree(my_tree,
          max_depth=2,
          fontsize=10,
          feature_names=medical_dataset.columns[:-1]
         )
```

```
['allergies' 'common cold' 'covid19' 'flu']
```



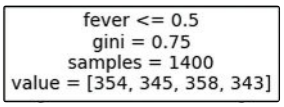
شكل 2.13: نموذج شجرة القرار لمجموعة بيانات medical_data (البيانات- الطبية) بعُمق مستويين

تُستخدَم دالة ()plot_tree لرسم وعرض شجرة القرار. ولعدم توفر مساحة كافية للعرض سيتم تمثيل المستويين الأوّلين فقط، بالإضافة إلى الجذر. يمكن ضبط هذا الرقم بسهولة باستخدام المُتغيِّر max_depth.

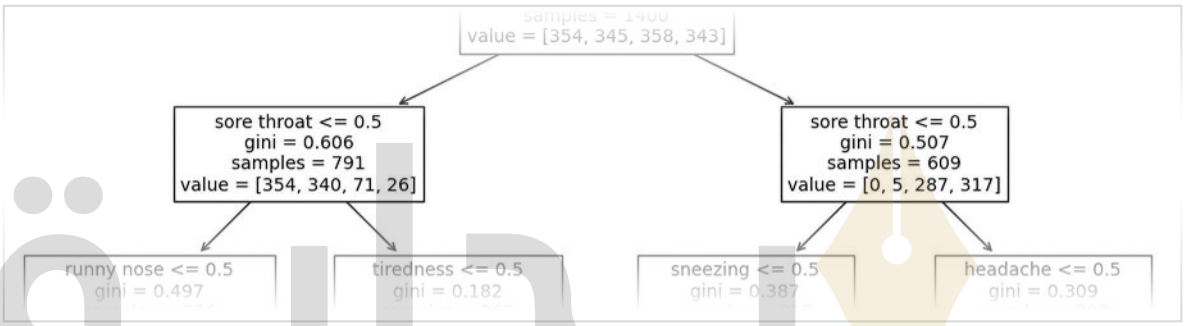
```
# plot the tree
plot_tree(my_tree,
          max_depth=2,
          fontsize=10
```

عُمق شجرة القرار.

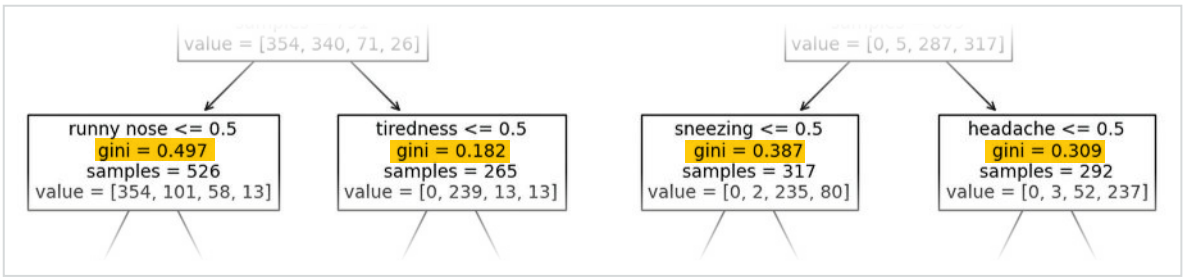
كل عُقدة في الشجرة تُمثِّل مجموعة فرعية من المرضى، فعلى سبيل المثال،



تُمثِّل عُقدة الجذر إجمالي عدد 1,400 مريض في مجموعة بيانات التدريب. من بينهم، 354، و345، و358، و343 شُخِّصوا بـ Allergies (الحساسية)، وCommon cold (نزلات البرد)، وCovid19 (كوفيد-19)، وFlu (الإنفلونزا)، على التوالي.



بُنيَّت الشجرة باستخدام نمط من الأعلى إلى الأسفل عبر **التفرُّع الثنائي** (Binary Splits). يَستنِد التفرُّع الأول إلى ما إذا كان المريض يُعاني من الحُمى أم لا. ونظرًا لأن كل خصائص الأعراض ثنائية، يكون التحقق a <=0.5 صحيحًا إذا لم يكن المريض يعاني من الأعراض. أما المرضى الذين لا يعانون من الحُمى (المسار الأيسر) يتفرَّعون مرة أخرى بناءً على ما إذا كانوا يعانون من التهاب الحلق أم لا. المرضى الذين لا يعانون من التهاب الحلق يتفرَّعون بناءً على ما إذا كانوا يعانون من رشح الأنف أم لا. في هذه المرحلة، تحتوي العُقدة على 526 حالة. تمّ تشخيص 354، و101، و58، و13 من بينهم بالحساسية، ونزلات البرد، وكوفيد-19، والإنفلونزا، على التوالي.



يستمر التفرُّع حتى تُحدِّد الخوارزمية الحالات التي انقسمت بالفعل إلى عُقد نقيَّة تمامًا. العُقدة النقيَّة بالكامل تحتوي على الحالات التي لها التشخيص نفسه. قِيَم مؤشر gini (جيني) المُحدَّدة على كل عُقدة، تُمثِّل مؤشرات على مقياس جيني، وهي صيغة شهيرة تُستخدَم لتقييم درجة نقاء العُقدة.

يقيس مؤشر جيني (Gini Index) الشوائب بالعُقدة، وبالتحديد احتمالية تصنيف محتويات العُقدة بصورة خاطئة. يشير انخفاض مُعامِل جيني إلى ارتفاع درجة تأكُّد الخوارزمية من التصنيف.

ستَستخدِم الآن شجرة القرار للتنبؤ بالتشخيص الأكثر احتمالًا للمرضى في مجموعة الاختبار.

تُستخدَم مجموعة الاختبار لتقييم أداء النموذج. تَستنِد طريقة التقييم الدقيقة على ما إذا كان المقصود من المهمة **الانحدار** (Regression) أم **التصنيف** (Classification). في مثل مشكلات التصنيف المعروضة هنا، تُستخدَم طرائق التقييم الشهيرة مثل: حساب **دقة النموذج** (Model's Accuracy) و**مصفوفة الدقة** (Confusion Matrix).

- الدقة هي نسبة التنبؤات الصحيحة التي يقوم بها المُصنِّف. تَحقُق دقة عالية قريبة من 100% يعني أن معظم التنبؤات التي يقوم بها المُصنِّف صحيحة.
- مصفوفة الدقة هي جدول يقارن بين القيم الحقيقية (الفعلية) وبين التنبؤات التي يقوم بها المُصنِّف في مجموعة البيانات. يحتوي الجدول على صف واحد لكل قيمة صحيحة وعمود واحد لكل قيمة مُتوقَّعة. كل مُدخَل في المصفوفة يُمثِّل عدد الحالات التي لها قيم فعلية ومُتوقَّعة.

```
# functions used to evaluate a classifier
from sklearn.metrics import accuracy_score,confusion_matrix

# drop the diagnosis column to get only the symptoms
test_patient_symptoms=test_data.drop(columns=['diagnosis'])

# get the diagnosis column, to be used as the classification target
test_diagnoses=test_data['diagnosis']

# guess the most likely diagnoses
pred=my_tree.predict(test_patient_symptoms)

# print the achieved accuracy score
accuracy_score(test_diagnoses,pred)
```

```
0.8166666666666667
```

ستلاحظ أن شجرة القرار تُحقِّق دقة تصل إلى 81.6%، وهذا يعني أنه من بين 600 حالة تمّ اختبارها، شَخَّصت الشجرة 490 منها بشكل صحيح. يُمكنك كذلك طباعة مصفوفة الدقة للنموذج لتستعرض بشكل أفضل الأمثلة المُصنَّفة بشكل خاطئ.

```
confusion_matrix(test_diagnoses,pred)
```

```
array([[143,   3,   0,   0],
       [ 48,  98,   5,   4],
       [  2,   1, 127,  12],
       [  1,   3,  31, 122]])
```

	الحساسية المُتوقَّعة	نزلات البرد المُتوقَّعة	كوفيد19- المُتوقَّع	الإنفلونزا المُتوقَّعة
الحساسية الفعلية	**143**	3	0	0
نزلات البرد الفعلية	48	**98**	5	4
كوفيد-19 الفعلي	2	1	**127**	12
الإنفلونزا الفعلية	1	3	31	**122**

شكل 2.14: مصفوفة الدقة للحالات المُتوقَّعة والحالات الفعلية

الأرقام الواقعة في الخط القُطري (المُظللة باللون الوردي) تُمثِّل الحالات المُتوقَّعة بشكل صحيح، أما الأرقام التي تقع خارج الخط القُطري فتُمثِّل أخطاء النموذج.

على سبيل المثال، بالنظر إلى ترتيب التشخيصات الأربعة المُحتملة [Allergies (الحساسية)، Common cold (نزلات البرد)، Covid19 (كوفيد-19)، Flu (الإنفلونزا)]، توضح المصفوفة أن النموذج أخطأ في تصنيف 48 حالة من المُصابين بنزلات البرد بأنهم مصابون بالحساسية، كما أخطأ في تصنيف 31 حالة من المُصابين بالإنفلونزا بأنهم مصابون بكوفيد-19.

وعلى الرغم من أنّ هذا النموذج ليس مثاليًا، فمن المُثير للدهشة أنّه قادر على تحقيق مثل هذه الدرجة العالية من الدقة بتعلُّم مجموعة القواعد الخاصة به، دون الحاجة إلى قاعدة معرفة أنشئت يدويًا. بالإضافة إلى تحقيق مثل هذه الدقة دون محاولة ضبط مُتغيرات الأداء المتنوعة لـ DecisionTreeClassifier (مُصنِّف شجرة القرار). وبالتالي، يُمكن تحسين دقة النموذج لأفضل من ذلك. كما يُمكن تحسين النموذج بتجاوز قيود النموذج القائم على القواعد وتجربة أنواع مختلفة من خوارزميات تعلُّم الآلة. وستتعلّم بعض هذه الطرائق في الوحدة التالية.

تمرينات

1 اذكر بعض مزايا وعيوب الأنظمة القائمة على القواعد.

- يمكنها اتخاذ القرارات وحل المشكلات بسرعة وبدقة أفضل من البشر خاصة عندما يتعلق الأمر بالمهام التي تتطلب قدراً كبيراً من المعرفة أو البيانات
- تعمل هذه الأنظمة باستمرار دون تحيز أو أخطاء قد تؤثر في بعض الأحيان على اتخاذ القرار البشري

- تعمل هذه الأنظمة بكفاءة طالما كانت مدخلات المعرفة والقواعد جيدة وقد لا تستطيع التعامل مع المواقف التي تقع خارج نطاق خبراتها
- لا يمكنها التعلم أو التكيف بالطريقة نفسها مثل البشر وهذا يجعلها أقل قابلية للتطبيق على الأحداث المتغيرة حيث تتغير مدخلات البيانات والمنطق كثيراً بمرور الوقت

2 ما مزايا وعيوب الإصدار الأول؟

المزايا: يتميز بالبديهية والقابلة للتفسير ،كما يتضمن استخدام قاعدة المعرفة والقواعد في التشخيص الطبي دون تحيز أو انحراف عن الخط المعياري

العيوب: أن قاعدة ثلاثة أعراض على الأقل هي تمثيل مبسط للغاية لكيفية التشخيص الطبي على يد الخبير البشري وأن قاعدة المعرفة داخل الدالة تكون محددة بتعليمات برمجية ثابتة

3 أضف إلى المقطع البرمجي الخاص بالإصدار الأول لنظام قائم على القواعد مريضًا يُعاني من الأعراض التالية [Vomiting (القيء)، وAbdominal pain (آلام البطن)، وDiarrhea (الإسهال)، وFever (الحُمى)، وLower back pain (ألم بأسفل الظهر)]. ما التشخيص الطبي لحالة المريض؟ دَوِّن ملاحظاتك بالأسفل.

التشخيص هو:(التسمم الغذائي)

```
#New patient
 my-symptoms=['vomiting','abdominal pain','diarrhea',','fever
  'lower back pain']
 diagnosis=diagnose-v1(my-symptoms

print ('Most likely  diagnosis :',diagnosis
```

4 في الإصدار الثاني، كم عدد الأمراض المُوضحَّة في تشخيص كل مريض إذا غَيَّرت قيمة المُتغيِّر matching_symptoms_lower_bound إلى 2 و3 و4؟ عَدِّل المقطع البرمجي ثم دَوِّن ملاحظاتك.

المريض 1/ المريض 2:
قيمة المتغير 2:['نزلات البرد','الانفلونزا','الحساسية']
قيمة المتغير 3 :['نزلات البرد','الانفلونزا','الحساسية']
قيمة المتغير 4:['نزلات البرد']
المريض 3:
قيمة المتغير 2:['covid 19','flu']
قيمة المتغير 3:[]
قيمة المتغير 4:[]

5 في الإصدار الثالث، غَيِّر كلا الوزنين إلى 1 للمريضين الأول والثاني، تمامًا مثل المريض الثالث.

عَدِّل المقطع البرمجي ثم دَوِّن ملاحظاتك.

المريض 1 :(['covid 19','flu','allergies'],3)
المريض 2 :(['common cold','flu'],4)
المريض 3 :(['common cold','flu'],4)

6 صفْ بإيجاز كيف يُمكن تحسين كل إصدار بالنسبة للإصدار السابق له (الأول إلى الثاني، والثاني إلى الثالث، والثالث إلى الرابع).

رابط الدرس الرقمي

www.ien.edu.sa

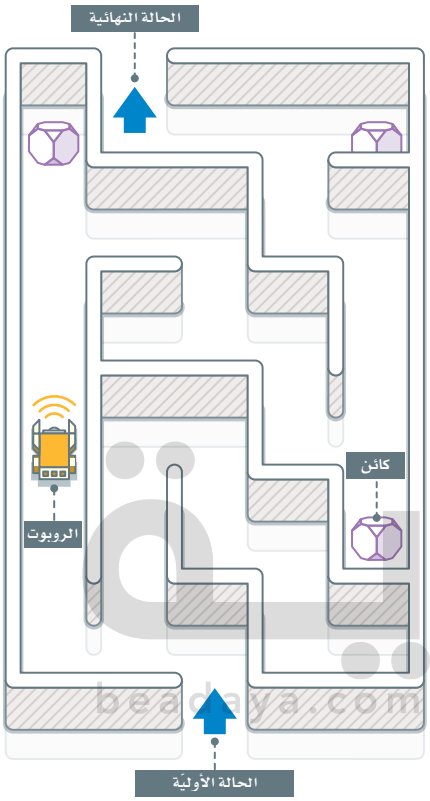
الدرس الرابع

خوارزميات البحث المستنيرة

تطبيقات خوارزميات البحث
Applications of Search Algorithms

خوارزميات البحث هي أحد المكونات الرئيسة لأنظمة الذكاء الاصطناعي، فباستخدامها يُمكن اكتشاف الاحتمالات المختلفة لإيجاد الحلول المناسبة للمشكلات المُعقدة في العديد من التطبيقات السائدة. وفيما يلي أمثلة على بعض تطبيقات خوارزميات البحث:

- **الروبوتية** (Robotics): قد يَستخدِم الروبوت خوارزمية البحث لتحديد طريقه عبر المتاهة أو لتحديد موقع أحد الكائنات في نطاق بيئته.
- **مواقع التجارة الإلكترونية** (E-commerce Websites): تَستخدِم مواقع التسوق عبر الإنترنت خوارزميات البحث لتُطابق بين استفسارات العملاء وبين المُنتجات المتوفرة، ولتصفية نتائج البحث وفق بعض المعايير مثل السعر، والعلامة التجارية، والتقييمات، واقتراح المُنتجات ذات الصلة.
- **منصّات مواقع التواصل الاجتماعي** (Social Media Platforms): تَستخدِم مواقع التواصل الاجتماعي خوارزميات البحث لعرض التدوينات، والأشخاص، والمجموعات للمُستخدِمين وفقًا للكلمات المفتاحية واهتمامات المُستخدِم.
- **تمكين الآلة من ممارسة الألعاب بمستوى عالٍ من المهارة** (Enabling a machine to play games at a high skill level): يَستخدِم الذكاء الاصطناعي خوارزمية البحث أثناء لعب الشطرنج أو قو (Go) لتقييم الحركات المختلفة واختيار الخطوات التي من المرجح أن تؤدي إلى الفوز.
- **نُظم الملاحة باستخدام مُحدِّد المواقع العالمي** (GPS Navigation Systems): تَستخدِم نُظم الملاحة القائمة على مُحدِّد المواقع العالمي خوارزميات البحث لتحديد أقصر وأسرع طريق بين موقعين، مع مراعاة بيانات حركة المرور في الوقت الحالي.
- **نُظم إدارة الملفات** (File Management Systems): تُستخدم خوارزميات البحث في نُظم إدارة الملفات لتحديد موقع الملفات باستخدام اسم، ومحتوى الملف، وبعض السمات الأخرى.



شكل 2.15: استخدام الروبوت خوارزمية البحث لتحديد طريقه

أنواع خوارزميات البحث وأمثلتها Types and Examples of Search Algorithms

هناك نوعان رئيسان من خوارزميات البحث وهما: **غير المُستنيرة** (Uninformed) و**المُستنيرة** (Informed).

خوارزميات البحث غير المستنيرة Uninformed Search Algorithms

خوارزميات البحث غير المُستنيرة، وتسمّى أيضًا: خوارزميات البحث العمياء، هي تلك التي لا تحتوي على معلومات إضافية حول حالات المشكلة بإستثناء المعلومات المستفادة من تعريف المشكلة. وتقوم هذه الخوارزميات بإجراء فحص شامل لمساحة البحث استنادًا إلى مجموعة من القواعد المُحدَّدة مُسبقًا. وتُعدُّ تقنيات **البحث بأولوية الاتساع** (BFS) و**البحث بأولوية العمق** (DFS) المُشار إليها في الدرس الثاني أمثلة على خوارزميات البحث غير المُستنيرة.

على سبيل المثال، تبدأ خوارزمية **البحث بأولوية العمق** (DFS) عند عُقدة الجذر بالشجرة أو المُخطَّط وتتوسَّع حتى تصل للعُقدة الأعمق التي لم تُفحَص. ويستمر الأمر بهذه الطريقة حتى تستنفد الخوارزمية مساحة البحث بأكملها بعد فحص كل العُقد المتاحة. ثم تُخرِج الحل الأمثل الذي وجدته أثناء البحث. فالحقيقة أن خوارزمية **البحث بأولوية العمق** (DFS) تَتبع دومًا هذه القواعد ولا يمكن ضبط استراتيجتها بصرف النظر عن نتائج البحث، وهذا ما يجعلها خوارزمية غير مُستنيرة.

ومثال آخر ملحوظ على هذا النوع من الخوارزميات هو خوارزمية **البحث بأولوية العمق التكراري المُتعمِّق** (Iterative Deepening Depth-First Search – IDDFS) التي يمكن اعتبارها مزيجًا بين خوارزميتي **البحث بأولوية العمق** (DFS) و**البحث بأولوية الاتساع** (BFS)، فهي تَستخدِم استراتيجة العُمق أولًا للبحث في جميع الخيارات الموجودة في النطاق الكامل بصورة متكررة حتى تصل إلى عُقدة مُحدَّدة.

خوارزميات البحث المستنيرة Informed Search Algorithms

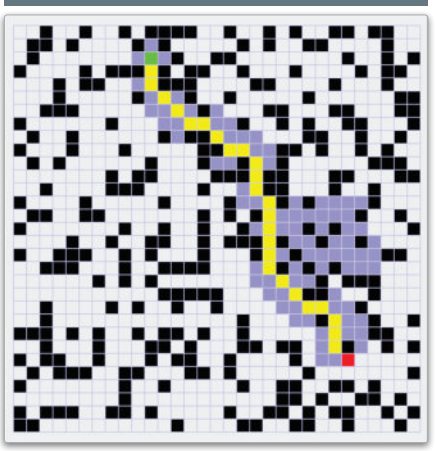
على النقيض من خوارزميات البحث غير المستنيرة، تَستخدِم خوارزميات البحث المُستنيرة المعلومات حول المشكلة ومساحة البحث لتوجيه عملية البحث. والأمثلة على هذه الخوارزميات تشمل:

- **خوارزمية البحث بأولوية الأفضل** (A* search) تَستخدِم دالة استدلالية لتقدير المسافة بين كل عُقدة من العُقد المُرشَّحة والعُقدة المُستهدَفة. ثم تُوسِّع العُقدة المُرشَّحة بالتقدير الأقل. إن فعَّالية خوارزمية **البحث بأولوية الأفضل** (A* search) مرتبطة بجودة دالتها الاستدلالية. على سبيل المثال، إذا كنت تضمن أن الاستدلال لن يتجاوز المسافة الفعلية إلى الهدف، فبالتالي سوف تعثر الخوارزمية على الحل الأمثل. بخلاف ذلك، قد لا يكون الحل الناتج من الخوارزمية هو الأفضل.
- **خوارزمية ديكسترا** (Dijkstra's Algorithm) تُوسِّع العُقدة بناء على أقصر مسافة فعلية إلى الهدف في كل خطوة. ولذلك، على النقيض من خوارزمية البحث بأولوية الأفضل، تحسب خوارزمية **ديكسترا** (Dijkstra) المسافة الفعلية ولا تَستخدِم التقديرات الاستدلالية. وبينما يجعل هذا خوارزمية ديكسترا أبطأ من خوارزمية البحث بأولوية الأفضل، إلا أن ذلك يعني ضمان العثور على الحل الأمثل دومًا (ممثلًا بالمسار الأقصر من البداية حتى الهدف).
- **خوارزمية تسلُّق التلال** (Hill Climbing) تبدأ بتوليد حل عشوائي، ثم تحاول تحسين هذا الحل بصورة متكررة بإجراء تغييرات بسيطة تُحسِّن من دالة استدلالية مُحدَّدة. وبالرغم من أن هذه المنهجية لا تضمن إيجاد الحل الأمثل، إلا أنها سهلة التنفيذ وتتميز بفعالية كبيرة عند تطبيقها على أنواع مُعينة من المشكلات.

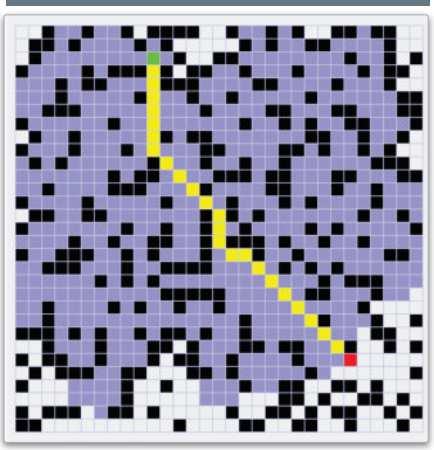
الدالة الاستدلالية (Heuristic function):

هي الدالة التي تُصنِّف البدائل في خوارزميات البحث عند كل مرحلة فرعية استنادًا إلى تقديرات استدلالية مبنية على البيانات المتوفرة لتحديد الفرع الذي ستسلكه.

خوارزمية البحث بأولوية الأفضل (A* search)



خوارزمية ديسكترا (Dijkstra's Algorithm)



الخلايا ذات اللون البنفسجي هي الخلايا التي تمّ فحصها، والخلية ذات اللون الأخضر هي موضع البدء، والخلية ذات اللون الاحمر هي موقع الهدف، بينما الخلايا ذات اللون الأصفر تمثل المسار الذي تم العثور عليه.

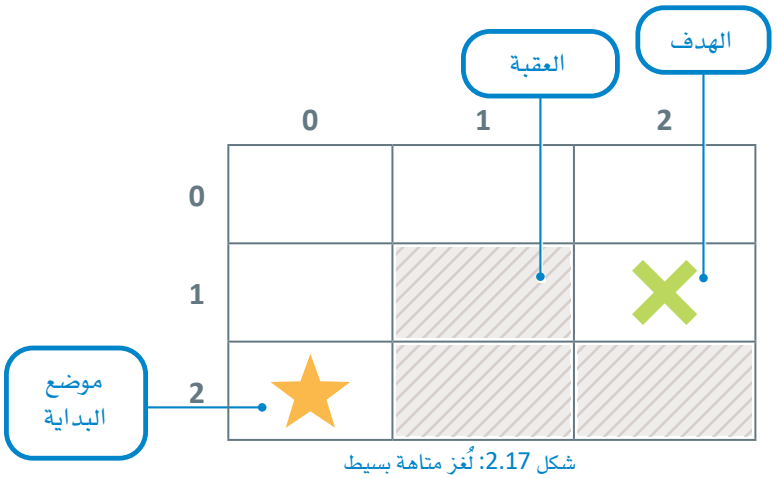
شكل 2.16: حل المتاهة نفسها باستخدام خوارزمية البحث بأولوية الأفضل وخوارزمية ديسكترا

في هذه الوحدة، ستشاهد بعض الأمثلة المرئية وتطبيقات البايثون على خوارزمية **البحث بأولوية الاتساع** (BFS) وخوارزمية **البحث بأولوية الأفضل** (A* search) لمعرفة الاختلافات بين خوارزميتي البحث المُستنيرة وغير المُستنيرة.

إنشاء ألغاز المتاهة بواسطة البايثون
Creating Maze Puzzles in Python

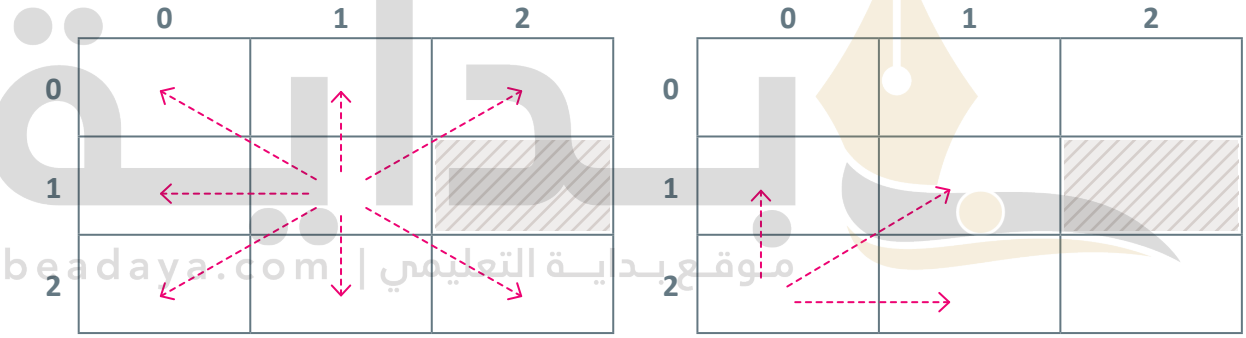
تُعرَّف المتاهة في صورة إطار شبكي 3×3.

يُحدَّد موضع البداية بنجمة في أسفل يسار المتاهة. الهدف هو الوصول إلى الخلية المُستهدَفة المُحدَّدة بالعلامة X، ويمكن للّاعب الانتقال إلى أي خلية فارغة مجاورة لموقعه الحالي.



شكل 2.17: لُغز متاهة بسيط

تكون الخلية فارغة إذا لم تحتوي على عائق. على سبيل المثال، المتاهة الموضَّحة في شكل 2.17 تحتوي على 3 خلايا تشغلُها **الحواجز** (Blocks). هذه الحواجز الملوّنة باللون الرمادي تُشكِّل عائقًا يجب على اللاعب تجاوزه للوصول إلى الهدف X، ويمكن للّاعب الانتقال بشكل أفقي أو رأسي أو قطري إلى أي خلية فارغة مجاورة لموقعه الحالي كما يظهر في شكل 2.18، على سبيل المثال:



شكل 2.18: يمكن للاعب الانتقال بشكل أفقي أو رأسي أو قطري إلى أي خلية فارغة مجاورة لموقعه الحالي

الهدف هو إيجاد المسار الأقصر والأقلّ عددًا لمرات فحص الخلايا. وبالرغم من أن المتاهة الصغيرة 3×3 قد تبدو بسيطة للّاعب البشري، إلا أنه يتوجب على الخوارزمية الذكية إيجاد حلول للتعامل مع المتاهات الكبيرة والمُعقدة للغاية، مثل: متاهة 10.000×10.000 التي تحتوي على ملايين الحواجز المُوزَّعة في أشكال مُعقَّدة ومتنوعة.

يمكن استخدام المقطع البرمجي التالي بلغة البايثون لإنشاء مجموعة بيانات تُصوّر المثال المُوضَّح في الشكل 2.18.

```
import numpy as np

# create a numeric 3 x 3 matrix full of zeros.
small_maze=np.zeros((3,3))

# coordinates of the cells occupied by blocks
blocks=[(1, 1), (2, 1), (2, 2)]

for block in blocks:
    # set the value of block-occupied cells to be equal to 1
    small_maze[block]=1

small_maze
```

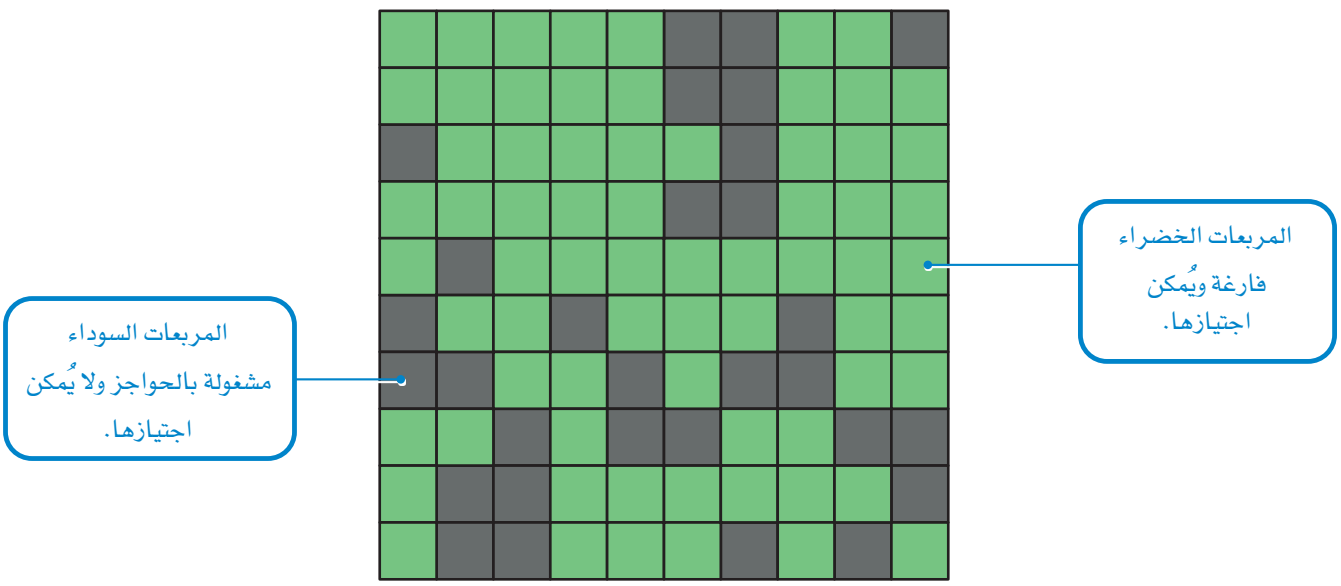
```
array([[0., 0., 0.],
       [0., 1., 0.],
       [0., 1., 1.]])
```

في هذا التمثيل الرقمي للمتاهة، تُمثَّل الخلايا الفارغة **بالأصفار** (Zeros) والمشغولة **بالآحاد** (Ones). يمكن تحديث المقطع البرمجي نفسه بسهولة لإنشاء متاهات كبيرة ومُعقدة للغاية، مثل:

```
import random

random_maze=np.zeros((10,10))

# coordinates of 30 random cells occupied by blocks
blocks=[(random.randint(0,9),random.randint(0,9)) for i in range(30)]

for block in blocks:
    random_maze[block]=1
```

تُستخدَم الدالة التالية لتمثيل المتاهة:

```
import matplotlib.pyplot as plt # library used for visualization

def plot_maze(maze):
    ax = plt.gca()          # create a new figure
    ax.invert_yaxis()       # invert the y-axis to match the matrix
    ax.axis('off')          # hide the axis labels
    ax.set_aspect('equal') # make sure the cells are rectangular

    plt.pcolormesh(maze, edgecolors='black', linewidth=2,cmap='Accent')
    plt.show()

plot_maze(random_maze)
```

شكل 2.19: تمثيل متاهة 10×10 باستخدام حواجز عشوائية

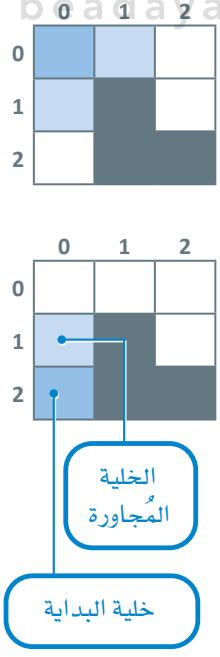
يُمكن استخدام الدالة التالية لاستدعاء قائمة تحتوي على كل الخلايا الفارغةِ والمُجاورة لخلية مُحدَّدة في أي متاهة:

```
def get_accessible_neighbors(maze:np.ndarray, cell:tuple):

    # list of accessible neighbors, initialized to empty
    neighbors=[]

    x,y=cell

    # for each adjacent cell position
    for i,j in [(x-1,y-1),(x-1,y),(x-1,y+1),(x,y-1),(x,y+1),(x+1,y-1),(x+1,y),
(x+1,y+1)]:

        # if the adjacent cell is within the bounds of the grid and is not occupied by a block
        if i>=0 and j>=0 and i<len(maze) and j<len(maze[0]) and
maze[(i,j)]==0:

            neighbors.append(((i,j),1))

    return neighbors
```

x-1, y-1	x-1, y	x-1, y+1
x, y-1	**x, y**	x, y+1
x+1, y-1	x+1, y	x+1, y+1

يفترض هذا التطبيق أن كل عملية انتقال من خلية إلى أخرى مجاورة سواءً أفقيًا أو رأسيًا أو قطريًا يتم بتكلفة مقدارها وِحدَةٌ وَاحِدَة فقط. سيتم إعادة النظر في هذه الفرضية في وقت لاحق من هذا الدرس بعرض حالات أكثر تعقيدًا مع شروط انتقال مُتغيرة.

تَستخدِم كل خوارزميات البحث دالة ()get_accessible_neighbors في محاولة حل المتاهة. في الأمثلة التالية تُستخدَم المتاهة 3×3 المُصمَّمة بالأعلى للتحقق من أن الدالة تستدعي الخلية الصحيحة الفارغة والمجاورة للخلية المُحدَّدة.



شكل 2.20: الخلايا المجاورة

```
# this cell is the northwest corner of the grid and has only 2 accessible neighbors
get_accessible_neighbors(small_maze, (0,0))
```

```
[((0, 1), 1), ((1, 0), 1)]
```

```
# the starting cell (in the southwest corner) has only 1 accessible neighbor
get_accessible_neighbors(small_maze, (2,0))
```

```
[((1, 0), 1)]
```

بعد أن تعلّمت كيفية إنشاء المتاهات، وكذلك استدعاء الخلايا المجاورة لأي خلية في المتاهة، فإنّ الخطوة التالية هي تطبيق خوارزميات البحث التي يمكنها حل المتاهة من خلال إيجاد المسار الأقصر من خلية البداية إلى خلية الهدف المُحدَّدة.

استخدام خوارزمية البحث بأولوية الاتساع في حل ألغاز المتاهة
Using BFS to Solve Maze Puzzles

تَستخدِم دالة bfs_maze_solver( ) المُشار إليها في هذا الجزء خوارزمية **البحث بأولوية الاتساع** (BFS) لحل ألغاز المتاهة باستخدام خلية البداية وخلية الهدف. يَستخدِم هذا النموذج دالة get_accessible_neighbors( ) المُحدَّدة بالأعلى لاستدعاء الخلايا المُجاورة التي يمكن فحصها عند أي نقطة أثناء البحث، و بمجرد عثور خوارزمية **البحث بأولوية الاتساع** (BFS) على الخليّة الهدف، ستُستخدم دالة reconstruct_shortest_path( ) المُوضحة بالأسفل لإعادة بِناء المسار الأقصر واستدعائه، وذلك بتتبع المسار بصورة عكسية من خلية الهدف إلى خلية البداية:

```
def reconstruct_shortest_path(parent:dict, start_cell:tuple, target_cell:tuple):

    shortest_path = []

    my_parent=target_cell # start with the target_cell

    # keep going from parent to parent until the search cell has been reached
    while my_parent!=start_cell:

        shortest_path.append(my_parent) # append the parent

        my_parent=parent[my_parent] # get the parent of the current parent

    shortest_path.append(start_cell) # append the start cell to complete the path

    shortest_path.reverse() # reverse the shortest path

    return shortest_path
```

ستُستخدَم دالة reconstruct_shortest_path( ) أيضًا لإعادة بِناء الحل لخوارزمية **البحث بأولوية الأفضل** (A* search) المُشار إليها سلفًا في هذا الدرس. وبالنظر إلى تعريف الدالتين get_accessible_ neighbors( ) و reconstruct_shortest_path( ) المُساعدتين، يُمكن تنفيذ دالة bfs_maze_solver( ) على النحو التالي:

```
from typing import Callable # used to call a function as an argument of another function

def bfs_maze_solver(start_cell:tuple,
                    target_cell:tuple,
                    maze:np.ndarray,
                    get_neighbors: Callable,
                    verbose:bool=False): # by default, suppresses descriptive output text

    cell_visits=0 # keeps track of the number of cells that were visited during the search
    visited = set() # keeps track of the cells that have already been visited
    to_expand = [] # keeps track of the cells that have to be expanded

    # add the start cell to the two lists
    visited.add(start_cell)
    to_expand.append(start_cell)
    # remembers the shortest distance from the start cell to each other cell
    shortest_distance = {}
    # the shortest distance from the start cell to itself, zero
```

```
    shortest_distance[start_cell] = 0

    # remembers the direct parent of each cell on the shortest path from the start_cell to the cell
    parent = {}
    #the parent of the start cell is itself
    parent[start_cell] = start_cell

    while len(to_expand)>0:

        next_cell = to_expand.pop(0) # get the next cell and remove it from the expansion list

        if verbose:
            print('\nExpanding cell', next_cell)

        # for each neighbor of this cell
        for neighbor,cost in get_neighbors(maze, next_cell):

            if verbose:
                print('\tVisiting neighbor cell',neighbor)

            cell_visits+=1

            if neighbor not in visited: # if this is the first time this neighbor is visited

                visited.add(neighbor)
                to_expand.append(neighbor)
                parent[neighbor]= next_cell
                shortest_distance[neighbor]=shortest_distance[next_cell]+cost

                # target reached
                if neighbor==target_cell:
                    # get the shortest path to the target cell, reconstructed in reverse.
                    shortest_path = reconstruct_shortest_path(parent,
                                                              start_cell, target_cell)

                    return shortest_path, shortest_distance[target_cell],cell_visits

            else: # this neighbor has been visited before

                # if the current shortest distance to the neighbor is longer than the shortest
                # distance to next_cell plus the cost of transitioning from next_cell to this neighbor
                if shortest_distance[neighbor]>shortest_distance[next_cell]
                                                                    +cost:

                    parent[neighbor]=next_cell
                    shortest_distance[neighbor]=shortest_distance[next_cell]+cost

    # search complete but the target was never reached, no path exists
    return None,None,None
```

تتّبع الدالة منهجية **البحث بأولوية الاتساع** (BFS) للبحث في كل الخيارات في العُمق الحالي قبل الانتقال إلى مستوى العُمق التالي، وتَستخدم هذه المنهجية مجموعة واحدة تُسمّى visited وقائمة تُسمى to_expand.

تتضمن المجموعة الأولى كل الخلايا التي فُحِصَت مرة واحدة على الأقل من قبل الخوارزمية. بينما تتضمن القائمة الثانية كل الخلايا التي لم تُوَسَّع بعد، مما يعني أن الخلايا المُجاورة لم تُفحص بعد. تَستخدم الخوارزمية كذلك قاموسين shortest_distance وparent، يحفظ الأوّل منهما طول المسار الأقصر من خلية البداية إلى كل خلية أخرى، بينما يحفظ الثاني عُقدة الخلية الأصل في المسار الأقصر.

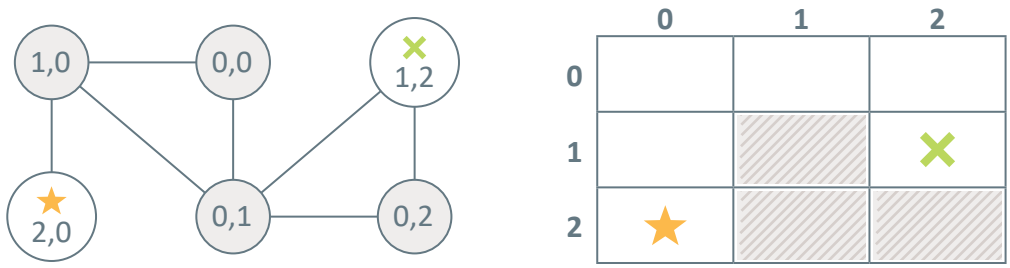
بمجرد الوصول إلى الخلية الهدف وانتهاء البحث، سيُخزِّن المتغيِّر shortest_distance[target_cell] طول الحل والذي يمثل طول المسار الأقصر من البداية إلى الهدف.

يَستخدِم المقطع البرمجي التالي دالة bfs_maze_solver( ) لحل المتاهة الصغيرة 3×3 المُوضَّحة بالأعلى:

```
start_cell=(2,0) # start cell, marked by a star in the 3x3 maze
target_cell=(1,2) # target cell, marked by an "X" in the 3x3 maze

solution, distance, cell_visits=bfs_maze_solver(start_cell,
                                    target_cell,
                                    small_maze,
                                    get_accessible_neighbors,
                                    verbose=True)

print('\nShortest Path:', solution)
print('Cells on the Shortest Path:', len(solution))
print('Shortest Path Distance:', distance)
print('Number of cell visits:', cell_visits)
```

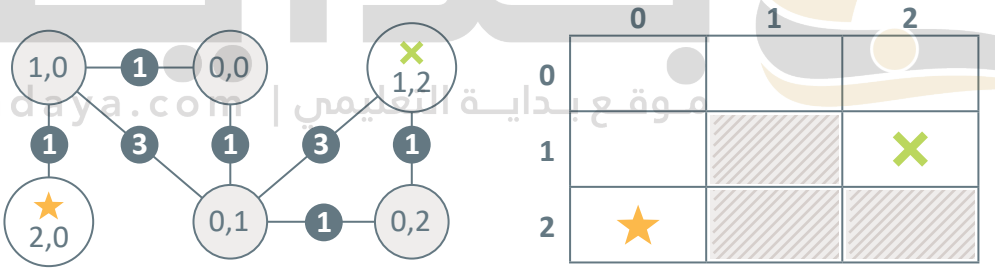
```
Expanding cell (2, 0)
      Visiting neighbor cell (1, 0)

Expanding cell (1, 0)
      Visiting neighbor cell (0, 0)
      Visiting neighbor cell (0, 1)
      Visiting neighbor cell (2, 0)

Expanding cell (0, 0)
      Visiting neighbor cell (0, 1)
      Visiting neighbor cell (1, 0)

Expanding cell (0, 1)
      Visiting neighbor cell (0, 0)
      Visiting neighbor cell (0, 2)
      Visiting neighbor cell (1, 0)
      Visiting neighbor cell (1, 2)

Shortest Path: [(2, 0), (1, 0), (0, 1), (1, 2)]
Cells on the Shortest Path: 4
Shortest Path Distance: 3
Number of cell visits: 10
```

تنجح خوارزمية **البحث بأولوية الاتساع** (BFS) في إيجاد المسار الأقصر بعد فحص 10 خلايا. يُمكن تصوير عملية البحث المطبّقة بخوارزمية **البحث بأولوية الاتساع** (BFS) بسهولة عند تصوير المتاهة بالتمثيل المُستنِد إلى مُخطَّط. المثال التالي يعرض متاهة 3×3 وتمثيلها بالمُخطَّط:



يتضمن تمثيل المُخطَّط عُقدة واحدة لكل خلية غير مشغولة. تُوضِّح القيمة على العُقد إحداثيات خلية المصفوفة المُقابلة. ستظهر حافة غير مُوجَّهة من عُقدة إلى أخرى في حال كانت الخلايا المُقابلة يُمكن الوصول إليها من خلال الانتقال من واحدة إلى الأخرى. إحدى الملاحظات المُهمّة حول خوارزمية **البحث بأولوية الاتساع** (BFS) هي أنه في حالة **المُخطَّطات غير الموزونة** (Unweighted Graphs) يكون المسار الأول الذي تُحدِّده الخوارزمية بين خلية البداية وأي خلية أخرى هو المسار الذي يتضمن أقل عدد من الخلايا التي تمّ فحصها. وهذا يعني أنه إذا كانت كلّ الحواف في المُخطَّط لها الوزن نفسه، أي كان لكلّ الانتقالات من خلية إلى أخرى التكلفة نفسها، فإنّ المسار الأول الذي تُحدِّده الخوارزمية إلى عُقدة مُحدَّدة يكون هو المسار الأقصر إلى تلك العُقدة. ولهذا السبب، تتوقف دالة bfs_( ) maze_solver عن البحث، وتَعرض نتيجة المرة الأولى التي فَحصَت فيها العُقدة المُستهدَفة.

ومع ذلك، لا يمكن تطبيق هذه المنهجية على **المُخطَّطات الموزونة** (Weighted Graphs). المثال التالي يوضِّح **إصدارًا موزونًا** (Weighted Version) لتمثيل مُخطَّط متاهة 3×3:



شكل 2.21: المتاهة ومُخطَّطها الموزون

في هذا المثال، يكون وزن كل الحواف المُقابلة للحركات الرأسية أو الأفقية (جنوبًا، شمالًا، غربًا، شرقًا) يساوي 1.

ومع ذلك، يكون وزن كل الحواف المُقابلة للحركات القُطرية (جنوبية غربية، جنوبية شرقية، شمالية غربية، شمالية شرقية) يساوي 3. في هذه الحالة الموزونة، سيكون المسار الأقصر هو [(2,0)، (1,0)، (0,0)، (0,1)، (0,2)، (1,2)]، بمسافة إجمالية: 1+1+1+1+1=5.

يمكن ترميز هذه الحالة الأكثر تعقيدًا باستخدام الإصدار الموزون من الدالة get_accessible_neighbors( ) المُوضَّحة بالأسفل.

```
def get_accessible_neighbors_weighted(maze:np.ndarray,
                                      cell:tuple,
                                      horizontal_vertical_weight:float,
                                      diagonal_weight:float):
```

```
    neighbors=[]
    x,y=cell

    for i,j in [(x-1,y-1), (x-1,y+1), (x+1,y-1), (x+1,y+1)]: # for diagonal neighbors

        # if the cell is within the bounds of the grid and it is not occupied by a block
        if i>=0 and j>=0 and i<len(maze) and j<len(maze[0]) and maze[(i,j)]==0:

            neighbors.append(((i,j), diagonal_weight))

    for i,j in [(x-1,y), (x,y-1), (x,y+1), (x+1,y)]: # for horizontal and vertical neighbors

        if i>=0 and j>=0 and i<len(maze) and j<len(maze[0]) and maze[(i,j)]==0:

            neighbors.append(((i,j), horizontal_vertical_weight))

    return neighbors
```

تسمح الدالة للمُستخدِم بتعيين وزن مُخصّص للحركات الأفقية و الحركات الرأسية، وكذلك وزن مُخصّص مختلف للحركات القُطرية. إذا استُخدِم **الإصدار الموزون** (Weighted Version) المُشار إليه بواسطة **أداة الحل في البحث بأولوية الاتساع** (BFS solver)، فإنَّ النتائج ستكون كما يلي:

```
from functools import partial

start_cell=(2,0)
target_cell=(1,2)
horz_vert_w=1 # weight for horizontal and vertical moves
diag_w=3 # weight for diagonal moves

solution, distance, cell_visits=bfs_maze_solver(start_cell,
                                    target_cell,
                                    small_maze,
                                    partial(get_accessible_neighbors_weighted,
                                           horizontal_vertical_weight=horz_vert_w,
                                            diagonal_weight=diag_w),
                                    verbose=False)

print('\nShortest Path:', solution)
print('Cells on the Shortest Path:', len(solution))
print('Shortest Path Distance:', distance)
print('Number of cell visits:', cell_visits)
```

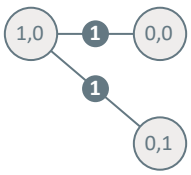
```
Shortest Path: [(2, 0), (1, 0), (0, 1), (1, 2)]
Cells on the Shortest Path: 4
Shortest Path Distance: 7
Number of cell visits: 6
```

وكما هو مُتوقَّع، أخطأت أداة **الحل في البحث بأولوية الاتساع** (BFS solver) في عرض المسار السابق نفسه بالضبط، على الرغم من أن التكلفة تساوي 7، ومن الواضح أنه ليس المسار الأقصر. ويرجع ذلك إلى الطبيعة غير المستنيرة لخوارزمية **البحث بأولوية الاتساع** (BFS)، حيث لا تأخذ الخوارزمية الأوزان بعين الاعتبار عند تحديد الخلية المُقرَّر توسيعها في الخطوة التالية؛ لأنها تُطبِّق ببساطة منهجية البحث بالعرض نفسِها والتي تؤدي إلى المسار نفسه الذي وجدته الخوارزمية في **الإصدار غير الموزون** (Unweighted Version) من المتاهة. القسم التالي يصف طريقة معالجة نقطة الضعف هذه باستخدام خوارزمية **البحث بأولوية الأفضل** (A* search)، وهي خوارزمية مُستنيرة وأكثر ذكاءً تضبط سلوكها وفقًا للأوزان المُحدَّدة، وبالتالي يُمكنها حل المتاهات باستخدام **الانتقالات الموزونة** (Weighted Transitions) و**الانتقالات غير الموزونة** (Unweighted Transitions).

استخدام خوارزمية البحث بأولوية الأفضل في حل ألغاز المتاهة
Using A* Search to Solve Maze Puzzles

كما في خوارزمية **البحث بأولوية الاتساع** (BFS)، تَفحَص خوارزمية **البحث بأولوية الأفضل** (A* search) خلية واحدة في كل مرة بفحص كل خلية مجاورة يمكن الوصول إليها. فبينما تستخدم خوارزمية **البحث بأولوية الاتساع** (BFS) منهجية بحث عمياء بأولوية العرض لتحديد الخلية التالية التي ستَفحَصها، تَفحَص خوارزمية **البحث بأولوية الأفضل** (A* search) الخلية التي يكون بينها وبين الخلية المُستهدَفة أقصر مسافة محسوبة بواسطة **الدالة الاستدلالية** (Heuristic Function). يعتمد التعريف الدقيق للدالة الاستدلالية على التطبيق. في حالة ألغاز المتاهة، توفِّر الدالة الاستدلالية تقديرًا دقيقًا لمدى قُرب الخلية المُرشَّحة إلى الخلية المُستهدَفة. يضمن الاستدلال المُطبَّق عدم **المبالغة في تقدير** (Overestimate) المسافة الفعلية إلى الخلية المُستهدَفة مثل: عرض مسافة تقديرية أكبر من المسافة الحقيقية إلى الهدف، وبالتالي سوف تُحدِّد الخوارزمية أقصر مسار مُحتمل لكل من المُخطَّطين **الموزون** (Weighted) و**غير الموزون** (Unweighted). إذا كان الاستدلال يُبالِغ في بعض الأحيان في تقدير المسافة، ستُقدِّم خوارزمية **البحث بأولوية الأفضل** (A* search) حلًا، ولكن قد لا يكون الأفضل. الاستدلال المُحتمل الأبسط الذي لن يؤدي إلى المُبالغة في تقدير المسافة هو دالة بسيطة تعطي دومًا مسافة تقديرية قدرها وِحدَة واحدة.


```
def constant_heuristic(candidate_cell:tuple, target_cell:tuple):

    return 1
```



شكل 2.22: الاستدلال الثابت

على الرغم من أن هذا الاستدلال شديد التفاؤل، إلا أنه لن يُقدم أبدًا تقديرًا أعلى من المسافة الحقيقية، وبالتالي سيؤدي إلى أفضل حل ممكن. سيتم تقديم استدلال متطور يُمكنه العثور على أفضل حل بشكلٍ سريع في هذا القسم لاحقًا.

تَستخدِم الدالة التالية دالة استدلالية معطاة للعثور على الخلية التي يجب توسيعها بعد ذلك:

```
def get_best_candidate(expansion_candidates:set,
                       shortest_distance:dict,
                       heuristic:Callable):

    winner = None
    # best (lowest) distance estimate found so far. Initialized to a very large number
    best_estimate= sys.maxsize

    for candidate in expansion_candidates:

        # distance estimate from start to target, if this candidate is expanded next
```

```
        candidate_estimate=shortest_distance[candidate]+heuristic(candidate,target_cell)
         if candidate_estimate < best_estimate:

             winner = candidate
             best_estimate=candidate_estimate

    return winner
```

يَستخدِم التطبيق المُشار إليه بالأعلى حلقة التكرار For لفحص الخلايا المُرشَّحة في المجموعة وتحديد الأفضل. ولِتطبيق أكثر كفاءة، قد يُستخدم **طابور الأولوية** (Priority Queue) في تحديد المُرشَّح الأفضل دون الحاجة إلى فحص كل المُرشَّحِين بصورة متكررة. تُستخدَم دالة get_best_candidate() كدالةٍ مُساعدة بواسطة دالة astar_maze_solver() المُوضَّحة فيما يلي. وبالإضافة إلى الدالة الاستدلالية، يَستخدِم هذا التطبيق كذلك الدالتين المُساعدتين get_accessible_neighbors_weighted() و reconstruct_shortest_path() المُشار إليهما في القسم السابق.

```
import sys

def astar_maze_solver(start_cell:tuple,
                      target_cell:tuple,
                      maze:np.ndarray,
                      get_neighbors: Callable,
                      heuristic:Callable,
                      verbose:bool=False):

    cell_visits=0

    shortest_distance = {}
    shortest_distance[start_cell] = 0

    parent = {}
    parent[start_cell] = start_cell

    expansion_candidates = set([start_cell])

    fully_expanded = set()

    # while there are still cells to be expanded
    while len(expansion_candidates) > 0:

        best_cell = get_best_candidate(expansion_candidates,shortest_distance,heuristic)

         if best_cell == None:  break

         if verbose: print('\nExpanding cell', best_cell)

         # if the target cell has been reached, reconstruct the shortest path and exit
         if best_cell == target_cell:
```

```
            shortest_path=reconstruct_shortest_path(parent,start_cell,target_cell)

            return shortest_path, shortest_distance[target_cell],cell_visits

        for neighbor,cost in get_neighbors(maze, best_cell):

            if verbose: print('\nVisiting neighbor cell', neighbor)

            cell_visits+=1

            # first time this neighbor is reached
            if neighbor not in expansion_candidates and neighbor not in fully_expanded:

                expansion_candidates.add(neighbor)

                parent[neighbor] = best_cell # mark the best_cell as this neighbor's parent

                # update the shortest distance for this neighbor
                shortest_distance[neighbor] = shortest_distance[best_cell] + cost

            # this neighbor has been visited before, but a better (shorter) path to it has just been found
            elif shortest_distance[neighbor] > shortest_distance[best_cell] + cost:

                shortest_distance[neighbor] = shortest_distance[best_cell] + cost

                parent[neighbor] = best_cell

                if neighbor in fully_expanded:

                    fully_expanded.remove(neighbor)

                    expansion_candidates.add(neighbor)

        # all neighbors of best_cell have been inspected at this point
        expansion_candidates.remove(best_cell)

        fully_expanded.add(best_cell)

    return None, None, None   # no solution was found
```


وكما الحال في الدالة ( )bfs_maze_solver، تَستخدِم الدالة المُوضَّحة بالأعلى كذلك القاموسين shortest_distance وparent لحفظ طول المسار الأقصر من خلية البداية إلى أي خلية أخرى، وحفظ عُقدة أصل الخلية في هذا المسار الأقصر.

ورغم ذلك، تَتبع الدالة ( )astar_maze_solve منهجية مختلفة لفحص الخلايا وتوسيعها، فهي تَستخدِم expansion_candidates لتتبُّع كل الخلايا التي يمكن توسيعها بعد ذلك. في كل تكرار، تُستخدَم دالة ( )get_best_candidate لتحديد أيٍّ من الخلايا المُرشَّحة ستُوسِّعُها بعد ذلك.

بعد اختيار الخليةِ المرشحةِ best_cell، تُستخدَم حلقة التكرار For لفحص كل الخلايا المجاورة لها. إذا كانت الخلية المجاورة تُفحَص للمرة الأولى، فستصبح best_cell عُقدة الأصل للخلية المجاورة في المسار الأقصر.

يحدث الأمر نفسه إذا تم فحص الدالة المجاورة من قبل، ولكن فقط إذا كان المسار إلى هذه الخلية المجاورة من خلال best_cell أقصر من المسار السابق. إذا عثرت الدالة بالفعل على مسار أفضل، فسيتعين على الخلية المجاورة العودة إلى مجموعة expansion_candidates لإعادة تقييم المسار الأقصر إلى الخلايا المجاورة لها. يُستخدم المقطع البرمجي التالي ( )astar_maze_solver لحل **الحالة غير الموزونة** (Unweighted Case) للُغز المتاهة 3x3:

```
start_cell=(2,0)
target_cell=(1,2)

solution, distance, cell_visits=astar_maze_solver(start_cell,
                                    target_cell,
                                    small_maze,
                                    get_accessible_neighbors,
                                    constant_heuristic,
                                    verbose=False)

print('\nShortest Path:', solution)
print('Cells on the Shortest Path:', len(solution))
print('Shortest Path Distance:', distance)
print('Number of cell visits:', cell_visits)
```

```
Shortest Path: [(2, 0), (1, 0), (0, 1), (1, 2)]
Cells on the Shortest Path: 4
Shortest Path Distance: 3
Number of cell visits: 12
```

ستبحث أداة **الحل في البحث بأولوية الأفضل** (A* search solver) عن المسار المُحتمل الأقصر والأفضل بعد فحص 12 خلية. وهذا أكثر قليلًا من أداة **الحل في البحث بأولوية الاتساع** (BFS solver) التي وجدت الحل بعد فحص 10 خلايا فقط. هذا يعود إلى بساطة الاستدلال الثابت المُستخدَم لإرشاد astar_maze_solver( ). وكما سيتضح لاحقًا في هذا القسم، يُمكن استخدام دالة استدلال أخرى لتمكين الخوارزمية من إيجاد الحل بشكلٍ أسرع.

الخطوة التالية هي تقييم ما إذا كانت خوارزمية **البحث بأولوية الأفضل** (A* search) قادرة على حل المتاهة الموزونة التي فشلت خوارزمية **البحث بأولوية الاتساع** (BFS) في العثور على أقصر مسار لها أم لا:

```
start_cell=(2,0)
target_cell=(1,2)

horz_vert_w=1  # weight for horizontal and vertical moves
diag_w=3  # weight for diagonal moves

solution, distance, cell_visits=astar_maze_solver(start_cell,
                                    target_cell,
                                    small_maze,
                                    partial(get_accessible_neighbors_weighted,
                                            horizontal_vertical_weight=horz_vert_w,
                                            diagonal_weight=diag_w),
                                    constant_heuristic,
                                    verbose=False)
```

```
print('\nShortest Path:', solution)
print('Cells on the Shortest Path:', len(solution))
print('Shortest Path Distance:', distance)
print('Number of cell visits:', cell_visits)
```

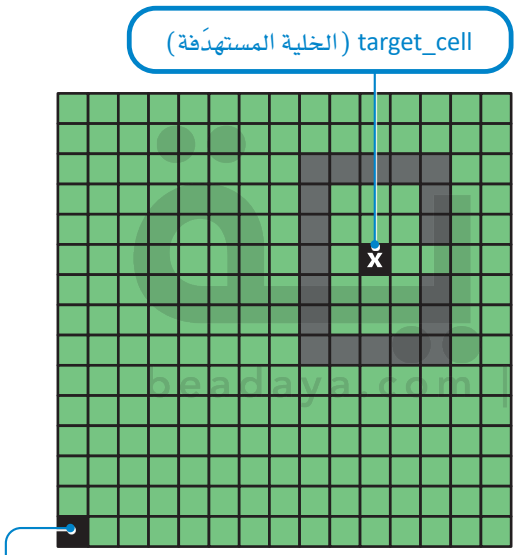
```
Shortest Path: [(2, 0), (1, 0), (0, 0), (0, 1), (0, 2), (1, 2)]
Cells on the Shortest Path: 6
Shortest Path Distance: 5
Number of cell visits: 12
```

تُوضِّح النتائج قدرة astar_maze_solver( ) على حل الحالة الموزونة بالعثور على المسار الأقصر المُحتمل [(2، 0)، (1، 0)، (0، 0)، (0، 1)، (0، 2)، (1، 2)] بتكلفة إجمالية قدرها 5. وهذا يوضِّح مزايا استخدام خوارزمية بحث مستنيرة، فهي تُمكِّنك من إيجاد الحل الأمثل باستخدام أبسط طريقة ممكنة.

المقارنة بين الخوارزميات Algorithm Comparison

الخطوة التالية هي المقارنة بين خوارزمية **البحث بأولوية الاتساع** (BFS) وخوارزمية **البحث بأولوية الأفضل** (A* search) في متاهة أكبر حجمًا وأكثر تعقيدًا. يُستخدَم المقطع البرمجي التالي بلغة البايثون لإنشاء تمثيل رقمي لهذه المتاهة:

```
big_maze=np.zeros((15,15))

# coordinates of the cells occupied by blocks
blocks=[(2,8), (2,9), (2,10), (2,11), (2,12),
        (8,8), (8,9), (8,10), (8,11), (8,12),
        (3,8), (4,8), (5,8), (6,8), (7,8),
        (3,12), (4,12), (6,12), (7,12)]

for block in blocks:
    # set the value of block-occupied cells to be equal to 1
    big_maze[block]=1
```



شكل 2.23: خلية البداية والخلية المستهدَفة للمتاهة

هذه المتاهة 15×15 تحتوي على قسم من الحواجز على شكل حرف C ينبغي على اللّاعب تجاوزها للوصول إلى الهدف المُحدَّد بالعلامة X. ثم تُستخدَم أداة **الحل في البحث بأولوية الاتساع** (BFS solver) وأداة **الحل في البحث بأولوية الأفضل** (A* search solver) لحل الإصدارات الموزونة وغير الموزونة من هذه المتاهة كبيرة الحجم:

الإصدار غير الموزون

```
start_cell=(14,0)
target_cell=(5,10)

solution_bfs_unw, distance_bfs_unw, cell_visits_bfs_unw=bfs_maze_solver(start_cell,
                                    target_cell,
                                    big_maze,
                                    get_accessible_neighbors,
```

```
                                        verbose=False)

print('\nBFS unweighted.')
print('\nShortest Path:', solution_bfs_unw)
print('Cells on the Shortest Path:', len(solution_bfs_unw))
print('Shortest Path Distance:', distance_bfs_unw)
print('Number of cell visits:', cell_visits_bfs_unw)

solution_astar_unw, distance_astar_unw, cell_visits_astar_unw=astar_maze_solver(
                                        start_cell,
                                        target_cell,
                                        big_maze,
                                        get_accessible_neighbors,
                                        constant_heuristic,
                                        verbose=False)

print('\nA* Search unweighted with a constant heuristic.')
print('\nShortest Path:', solution_astar_unw)
print('Cells on the Shortest Path:', len(solution_astar_unw))
print('Shortest Path Distance:', distance_astar_unw)
print('Number of cell visits:', cell_visits_astar_unw)
```

```
BFS unweighted.

Shortest Path: [(14, 0), (13, 1), (12, 2), (11, 3), (10, 4), (9, 5), (8,
6), (8, 7), (9, 8), (9, 9), (9, 10), (9, 11), (9, 12), (8, 13), (7, 13),
(6, 13), (5, 12), (4, 11), (5, 10)]
Cells on the Shortest Path: 19
Shortest Path Distance: 18
Number of cell visits: 1237

A* Search unweighted with a constant heuristic.

Shortest Path: [(14, 0), (13, 1), (12, 2), (11, 3), (10, 4), (10, 5), (10,
6), (9, 7), (9, 8), (10, 9), (9, 10), (9, 11), (9, 12), (8, 13), (7, 13),
(6, 13), (5, 12), (6, 11), (5, 10)]
Cells on the Shortest Path: 19
Shortest Path Distance: 18
Number of cell visits: 1272
```


الإصدار الموزون

```
start_cell=(14,0)
target_cell=(5,10)

horz_vert_w=1
diag_w=3

solution_bfs_w, distance_bfs_w, cell_visits_bfs_w=bfs_maze_solver(start_cell,
                                        target_cell,
```

```
                                    big_maze,
                                    partial(get_accessible_neighbors_weighted,
                                            horizontal_vertical_weight=horz_vert_w,
                                            diagonal_weight=diag_w),
                                    verbose=False)

print('\nBFS weighted.')
print('\nShortest Path:', solution_bfs_w)
print('Cells on the Shortest Path:', len(solution_bfs_w))
print('Shortest Path Distance:', distance_bfs_w)
print('Number of cell visits:', cell_visits_bfs_w)

solution_astar_w, distance_astar_w, cell_visits_astar_w=astar_maze_solver(start_cell,
                                    target_cell,
                                    big_maze,
                                    partial(get_accessible_neighbors_weighted,
                                            horizontal_vertical_weight=horz_vert_w,
                                            diagonal_weight=diag_w),
                                    constant_heuristic,
                                    verbose=False)

print('\nA* Search weighted with constant heuristic.')
print('\nShortest Path:', solution_astar_w)
print('Cells on the Shortest Path:', len(solution_astar_w))
print('Shortest Path Distance:', distance_astar_w)
print('Number of cell visits:', cell_visits_astar_w)
```

```
BFS weighted.

Shortest Path: [(14, 0), (14, 1), (14, 2), (13, 2), (13, 3), (12, 3), (12,
4), (11, 4), (11, 5), (10, 5), (10, 6), (9, 6), (9, 7), (9, 8), (9, 9), (9,
10), (9, 11), (9, 12), (9, 13), (8, 13), (7, 13), (6, 13), (5, 13), (5,
12), (4, 11), (5, 10)]
Cells on the Shortest Path: 26
Shortest Path Distance: 30
Number of cell visits: 1235

A* Search weighted with constant heuristic.

Shortest Path: [(14, 0), (13, 0), (12, 0), (11, 0), (10, 0), (9, 0), (9,
1), (9, 2), (9, 3), (9, 4), (9, 5), (9, 6), (9, 7), (9, 8), (9, 9), (9,
10), (9, 11), (9, 12), (9, 13), (8, 13), (7, 13), (6, 13), (5, 13), (5,
12), (5, 11), (5, 10)]
Cells on the Shortest Path: 26
Shortest Path Distance: 25
Number of cell visits: 1245
```

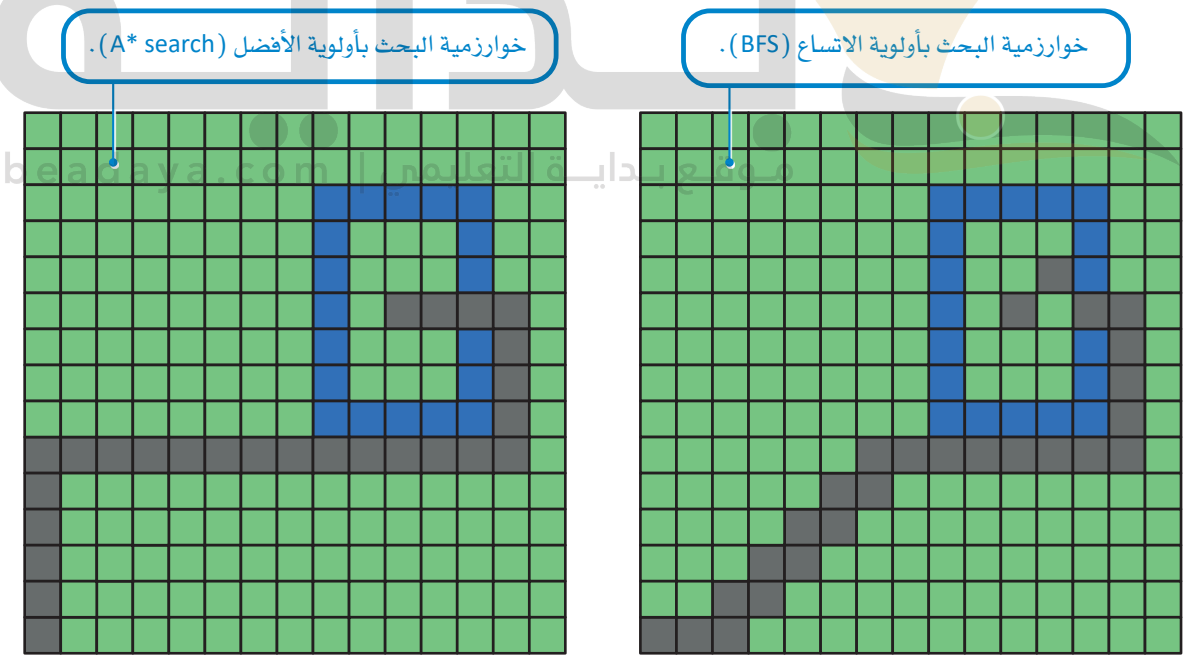
تتوافق النتائج مع تلك التي حصلت عليها في المتاهة الصغيرة وهي كالتالي:

- نجحت خوارزميّتا البحث بأولوية الاتساع (BFS) والبحث بأولوية الأفضل (A* search) في العثور على المسار الأقصر للإصدار غير الموزون.
- وجدت خوارزمية **البحث بأولوية الاتساع** (BFS) الحل بعد فحص عدد أقلّ من الخلايا وهو 1237 مقابل 1272 في خوارزمية **البحث بأولوية الأفضل** (A* search).
- فشلت خوارزمية **البحث بأولوية الاتساع** (BFS) في العثور على المسار الأقصر للإصدار الموزون، حيث عثرت على مسار بطول 30 وحدة.
- عثرت خوارزمية **البحث بأولوية الأفضل** (A* search) على المسار الأقصر للإصدار الموزون، حيث عثرت على مسار بطول 25 وحدة.

يُستخدَم المقطع التالي لتمثيل المسار الأقصر الذي وجدته الخوارزميتان؛ خوارزمية **البحث بأولوية الاتساع** (BFS) وخوارزمية **البحث بأولوية الأفضل** (A* search) للإصدار الموزون كالتالي:

```
maze_bfs_w=big_maze.copy()

for cell in solution_bfs_w:
    maze_bfs_w[cell]=2

plot_maze(maze_bfs_w)
```

```
maze_astar_w=big_maze.copy()

for cell in solution_astar_w:
    maze_astar_w[cell]=2

plot_maze(maze_astar_w)
```



شكل 2.24: مقارنة بين حلَّي خوارزميتي البحث بأولوية الاتساع والبحث بأولوية الأفضل

يؤكد التّمثيلان أن الطبيعة المُستنيرة لخوارزمية **البحث بأولوية الأفضل** (A* search) تسمح لها بتجنب الحركة القُطرية، لأن تكلفتها أعلى من الحركتين الأفقية والرأسية. ومن ناحية أخرى، تتجاهل خوارزمية **البحث بأولوية الأفضل** (BFS) غير المستنيرة تكلفة كل حركة وتُعطي حلًا أعلى تكلفة. وفيما يلي مقارنة عامة بين الخوارزميات المُستنيرة وغير المُستنيرة كما هو موضح في جدول 2.6:

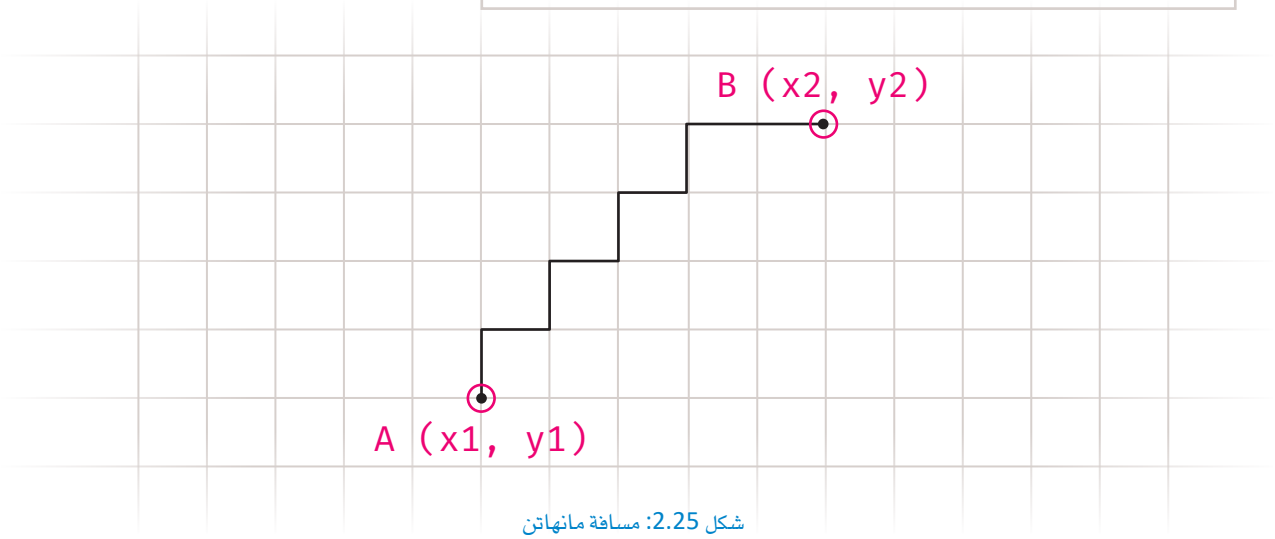
جدول 2.6: مقارنة بين الخوارزميات المُستنيرة وغير المُستنيرة

معايير المقارنة	المُستنيرة	غير المُستنيرة
التعقيد الحسابي (Computational Complexity)	أقل تعقيدًا.	أكثر تعقيدًا حسابيًا.
الكفاءة (Efficiency)	أسرع في عمليات البحث.	أبطأ من الخوارزميات المُستنيرة.
الأداء (Performance)	أفضل في حل مشكلات البحث واسع النطاق.	غير عملية لحل مشكلات البحث واسع النطاق.
الفعالية (Effectiveness)	تحقّقُ حلولا مُناسبةً بشكلٍ عام.	تُحقِق الحل الأمثل.

ومع ذلك، تُظهِر النتائج أن خوارزمية **البحث بأولوية الاتساع** (BFS) يمكنها العثور على الحل الأمثل بشكلٍ سريع بفحص عدد أقل من الخلايا في الحالة غير الموزونة. يمكن مُعالجة ذلك بتوفير استدلال أكثر ذكاءً لخوارزمية **البحث بأولوية الأفضل** (A* search). والاستدلال الشهير في التطبيقات المُستنِدة إلى المسافة هو **مسافة مانهاتن** (Manhattan Distance)، وهي مجموع الفروقات المُطلقة بين إحداثِيَّي نقطتين مُعطاتين. يوضِّح الشكل أدناه مثالًا على كيفية حساب مسافة مانهاتن:

مسافة مانهاتن Manhattan Distance

```
Manhattan (A, B) = |x1-x2| + |y1-y2|
```



شكل 2.25: مسافة مانهاتن

يمكن تطبيق هذا بسهولة في صورة دالة البايثون كما يلي:

```
def manhattan_heuristic(candidate_cell:tuple,target_cell:tuple):

    x1,y1=candidate_cell
    x2,y2=target_cell
    return  abs(x1 - x2) + abs(y1 - y2)
```

يُستخدَم المقطع البرمجي التالي لاختبار إمكانية استخدام هذا الاستدلال الذكي لدعم astar_maze_solver( ) في البحث بشكل أسرع في كلٍ من الحالات الموزونة وغير الموزونة:

```
start_cell=(14,0)
target_cell=(5,10)

solution_astar_unw_mn, distance_astar_unw_mn, cell_visits_astar_unw_mn=astar_
maze_solver(start_cell,
            target_cell,
            big_maze,
            get_accessible_neighbors,
            manhattan_heuristic,
            verbose=False)

print('\nA* Search unweighted with the Manhattan heuristic.')
print('\nShortest Path:', solution_astar_unw_mn)
print('Cells on the Shortest Path:', len(solution_astar_unw_mn))
print('Shortest Path Distance:', distance_astar_unw_mn)
print('Number of cell visits:', cell_visits_astar_unw_mn)

horz_vert_w=1 # weight for horizontal and vertical moves
diag_w=3  # weight for diagonal moves

solution_astar_w_mn, distance_astar_w_mn, cell_visits_astar_w_mn=astar_maze_
solver(start_cell,
       target_cell,
       big_maze,
       partial(get_accessible_neighbors_weighted,
               horizontal_vertical_weight=horz_vert_w,
               diagonal_weight=diag_w),
       manhattan_heuristic,
       verbose=False)

print('\nA* Search weighted with the Manhattan heuristic.')
print('\nShortest Path:', solution_astar_w_mn)
print('Cells on the Shortest Path:', len(solution_astar_w_mn))
print('Shortest Path Distance:', distance_astar_w_mn)
print('Number of cell visits:', cell_visits_astar_w_mn)
```

```
A* Search unweighted with the Manhattan heuristic.

Shortest Path: [(14, 0), (13, 1), (12, 2), (11, 3), (10, 4), (9, 5), (8,
6), (8, 7), (9, 8), (9, 9), (9, 10), (9, 11), (9, 12), (8, 13), (7, 13),
(6, 13), (5, 12), (5, 11), (5, 10)]
Cells on the Shortest Path: 19
Shortest Path Distance: 18
Number of cell visits: 865

A* Search weighted with the Manhattan heuristic.

Shortest Path: [(14, 0), (14, 1), (13, 1), (12, 1), (12, 2), (12, 3), (12,
4), (12, 5), (12, 6), (12, 7), (11, 7), (11, 8), (10, 8), (9, 8), (9, 9),
(9, 10), (9, 11), (9, 12), (9, 13), (8, 13), (7, 13), (6, 13), (5, 13), (5,
12), (5, 11), (5, 10)]
Cells on the Shortest Path: 26
Shortest Path Distance: 25
Number of cell visits: 1033
```

تؤكد النتائج أن استدلال **مسافة مانهاتن** (Manhattan Distance) يمكن استخدامه لدعم خوارزمية **البحث بأولوية الأفضل** (A* search) في العثور على المسارات الأقصر المُحتمَلة بفحص أقل عدد من الخلايا في كل من الحالات الموزونة وغير الموزونة. علمًا بأن استخدام هذا الاستدلال الأكثر ذكاءً يفحص عددًا أقل من الخلايا من ذلك المُستخدَم في خوارزمية **البحث بأولوية الاتساع** (BFS).

يُلخِّص جدول 2.7 النتائج حول مُتغيرات الخوارزميّات المختلفة في المتاهة الكبيرة:

جدول 2.7: مقارنة بين أداء الخوارزميات

	خوارزمية البحث بأولوية الاتساع (BFS)	خوارزمية البحث بأولوية الأفضل (A* search) بالاستدلال الثابت	خوارزمية البحث بأولوية الأفضل (A* search) باستدلال مانهاتن
الموزونة	المسافة=30، وفَحصَت 1235	المسافة=25، وفَحصَت 1245	المسافة=25، وفَحصَت 1033
غير الموزونة	المسافة=18، وفَحصَت1237	المسافة=18، وفَحصَت 1272	المسافة=18، وفَحصَت865

يُوضِّح الجدول مزايا استخدام الطَّرائق الأكثر ذكاءً لحل المشكلات المُستنِدة إلى البحث مثل تلك المُوضَّحة بهذا الدرس:

- التَحوُّل من خوارزمية **البحث بأولوية الاتساع** (BFS) غير الموزونة إلى خوارزمية **البحث بأولوية الأفضل** (A* search) الموزونة حَقَّق نتائجَ أفضل، كما أتاح إمكانية حل المشكلات الأكثر تعقيدًا.
- يُمكن تحسين ذكاء خوارزميات البحث المُستنيرة باستخدام دوال الاستدلال الأفضل التي تسمح لها بالعثور على الحل الأمثل بشكلٍ أسرع.

تمرينات

1 اذكر تطبيقين لخوارزميات البحث.

1-الروبوتية

2-مواقع التجارة االكترونية

2 حدِّد الاختلافات بين خوارزميات البحث المُستنيرة وغير المُستنيرة، ثم اذكر مثالًا على كل خوارزمية.

يمكن للطلبة الرجوع لصفحة 107 ، 108 من الكتاب لحل هذا السؤال

3 اشرح بإيجاز كيف تعمل خوارزمية البحث بأولوية الأفضل (A* search).

يمكن للطلبة الرجوع لصفحة 117 من الكتاب لحل هذا السؤال

4 عَدِّل المقطع البرمجي بتغيير الوزن القُطري (Diagonal Weight) من 3 إلى 1.5. ماذا تُلاحظ؟
هل يتغير المسار الأقصر في حالتي خوارزمية البحث بأولوية الاتساع (BFS) وخوارزمية البحث بأولوية الأفضل (A* search)؟

في المتاهة 3x3 يختصر المسار فقط في حالة خوارزمية البحث بأولوية الأفضل .
في المتاهة 15x15 يختصر المسار في كل من خوارزمية البحث بأولوية الاتساع وخوارزمية البحث بأولوية الأفضل

5 عَدِّل المقطع البرمجي بتبديل إحداثيات خلية البداية مع احداثيات الخلية المُستهدَفة. ماذا تُلاحظ؟
هل المسار هو نفسه كما كان سابقًا للحالات الموزونة من خوارزميتي البحث بأولوية الاتساع (BFS) والبحث بأولوية الأفضل (A* search)؟

في متاهة3x3 ، يكون المسار أطول ل BFS
بالنسبة لمتاهة 15x15 ،يكون لكل من BFS و A* مسارات مختلفة ولكن بنفس الطول كما كان من قبل .

المشروع

1. عدِّل المقطع البرمجي لخوارزمية **البحث بأولوية الاتساع** (BFS) وخوارزمية **البحث بأولوية الأفضل** (A* search) الموزونتين بتغيير الأوزان الأفقية والرأسية إلى 3 والأوزان القُطرية إلى 5. وكذلك عدِّل نقطة البداية إلى (2، 7).

2. ما المسار الجديد ذو المسافة الأقصر، وما عدد الخلايا التي فُحِصَت في الإصدارات غير الموزونة لخوارزميتي **البحث بأولوية الاتساع** (BFS) و**البحث بأولوية الأفضل** (A* search) باستخدام دالة الاستدلال الثابت؟ حَدِّد هذه القيم ودوِّن ملاحظاتك.

3. اتبع الخطوات نفسها للإصدارات الموزونة من خوارزميتي **البحث بأولوية الاتساع** (BFS) و**البحث بأولوية الأفضل** (A* search) باستخدام دالة الاستدلال الثابت.

4. كرِّر العملية للإصدارات غير الموزونة والموزونة من خوارزميتي **البحث بأولوية الاتساع** (BFS) و**البحث بأولوية الأفضل** (A* search) باستخدام دالة **استدلال مانهاتن** (Manhattan Heuristic).

ماذا تعلّمت

- استخدام الاستدعاء الذاتي لحل المشكلات.
- تطبيق خوارزميات اجتياز المُخطَّط المُتقدمة.
- تطبيق الأنظمة القائمة على القواعد البسيطة والمتقدمة.
- تصميم نموذج الذكاء الاصطناعي.
- قياس فعالية نموذج الذكاء الاصطناعي الذي صمّمتَه.
- استخدام خوارزميات البحث لمحاكاة حلّ مشكلات الحياة الواقعية.

المصطلحات الرئيسة

تدريب النموذج	Model Training
إيجاد المسار	Path Finding
الاستدعاء الذاتي	Recursion
الأنظمة القائمة على القواعد	Rule-Based Systems
دالة تسجيل النقاط	Scoring Function
خوارزميات البحث	Search Algorithms
البحث غير المُستنير	Uninformed Search
مُخطَّط غير موزون	Unweighted Graph
مُخطَّط موزون	Weighted Graph

البحث بأولوية الأفضل	A* Search
أداء الخوارزمية	Algorithm Performance
البحث بأولوية الاتساع	Breadth-First Search (BFS)
مصفوفة الدقة	Confusion Matrix
البحث بأولوية العمق	Depth-First Search (DFS)
دالة استدلالية	Heuristic Function
البحث المُستنير	Informed Search
قاعدة المعرفة	Knowledge Base
حل المتاهات	Maze Solving