**شرح خوارزمية البحث العرضي الأول**

**ما هي خوارزمية البحث العرضي**

خوارزمية البحث العرضي (Breadth-First Search) هي إحدى الخوارزميات الأساسية في علوم الحاسوب، تستخدم لاستكشاف جميع الرؤوس والروابط في بنية بيانات شجرية أو بيانية. على عكس DFS، تقوم BFS بزيارة جميع الجيران المباشرين لرأس ما قبل الانتقال إلى الجيل التالي من الجيران

**متى نستخدمها**

لإيجاد أقصر مسار بين نقطتين في رسم بياني غير موجه وغير مرجح: **البحث عن أقصر مسار**

لتقسيم الرسم البياني إلى مكونات متصلة. **الكشف عن مكونات متصلة:**

**حل المسائل المتعلقة بالبحث** :مثل مشكلة المتاهة

**كيف تعمل**

**البدء برأس:** نختار رأسًا بدءًا للبحث

الزيارة والوضع في المكدس**:** نضع الرأس الحالي في المكدس ونعلم بأنه تمت زيارته

اختيار جار**:** نختار أحد جيران الرأس الحالي الذي لم يسبق زيارته

التكرار: نكرر الخطوات 2 و 3 حتى لا يوجد جيران غير مزارين للرأس الحالي

الرجوع: إذا لم يكن هناك جيران غير مزارين، نخرج من الرأس الحالي ونعود إلى الرأس الذي سبق زيارته

التكرار: نكرر الخطوات 3 و 4 و 5 حتى يتم زيارة جميع الرؤوس المتصلة برأس البداية

**تطبيق** Python

from collections import deque

def bfs(graph, start):

visited = set() # مجموعة لتخزين الرؤوس التي تم زيارتها

queue = deque([start]) # قائمة الانتظار

while queue:

node = queue.popleft()

if node not in visited:

visited.add(node)

print(node, end=' ')

for neighbor in graph[node]:

queue.append(neighbor)

# مثال على استخدام الدالة:

graph = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

bfs(graph, 'A') # ستطبع: A B C D E F

.

**شرح الكود**

* **deque:** تستخدم هنا لتمثيل قائمة الانتظار (FIFO).
* **visited:** مجموعة لتتبع الرؤوس التي تم زيارتها.
* **queue:** قائمة الانتظار التي تحتوي على الرؤوس التي يجب زيارتها.
* **الحلقة while:** تستمر حتى تصبح قائمة الانتظار فارغة.
* **node = queue.popleft():** إزالة الرأس الأول من القائمة.
* **visited.add(node):** إضافة الرأس إلى مجموعة الزيارات.
* **for neighbor in graph[node]::** تكرار على جميع جيران الرأس الحالي.
* **queue.append(neighbor):** إضافة الجيران إلى قائمة الانتظار إذا لم يتم زيارتها من قبل.

**ملاحظات:**

* **تمثيل الرسم البياني:** نفس التمثيل المستخدم في DFS.
* **تعديل الدالة:** يمكن تعديل هذه الدالة لإيجاد أقصر مسار أو حل مسائل أخرى.
* **تعقيد زمني:** في أسوأ الحالات، يكون تعقيد الوقت الزمني لـ BFS هو O(V+E)، حيث V هو عدد الرؤوس وE هو عدد الحواف.

**مقارنة بين BFS و DFS:**

|  |  |  |
| --- | --- | --- |
| **الخاصية** | **BFS** | **DFS** |
| استكشاف | عرضي (مستوى بعد مستوى) | عمقي (أقصى عمق ثم العودة) |
| بنية البيانات | قائمة انتظار | مكدس |
| إيجاد أقصر مسار | فعال في الرسوم البيانية غير الموجهة وغير المرجحة | قد لا يكون فعالًا |
| الكشف عن الدورات | قد لا يكون فعالًا | فعال في الكشف عن دورات |

التصدير إلى "جداول بيانات Google"

**تطبيقات BFS:**

* **البحث في شبكات الإنترنت:** للعثور على صفحات الويب المتصلة بصفحة معينة.
* **تحليل شبكات التواصل الاجتماعي:** للعثور على الأصدقاء المشتركين بين المستخدمين.
* **حل مشاكل المتاهات:** للعثور على أقصر طريق للخروج من المتاهة.

## شرح خوارزمية البحث العميق أولاً (DFS) وكودها في بايثون

**ما هي خوارزمية البحث العميق DFS؟**

هي إحدى الخوارزميات الأساسية في علوم الحاسوب، تستخدم لاستكشاف جميع الرؤوس والروابط في بنية بيانات شجرية أو بيانية. تعمل الخوارزمية على فحص كل فرع على العمق الأقصى قبل الانتقال إلى الفرع التالي.

**متى نستخدم DFS؟**

* **البحث عن مسار:** لإيجاد مسار بين نقطتين في الرسم البياني.
* **الكشف عن الدورات:** للتحقق من وجود دورات في الرسم البياني.
* **تلوين الرؤوس:** لتقسيم الرسم البياني إلى مكونات متصلة.
* **حل المسائل المتعلقة بالبحث:** مثل مشكلة المتاهة.

**كيف تعمل DFS؟**

1. **البدء برأس:** نختار رأسًا بدءًا للبحث.
2. **الزيارة والوضع في المكدس:** نضع الرأس الحالي في المكدس ونعلم بأنه تمت زيارته.
3. **اختيار جار:** نختار أحد جيران الرأس الحالي الذي لم يسبق زيارته.
4. **التكرار:** نكرر الخطوات 2 و 3 حتى لا يوجد جيران غير مزارين للرأس الحالي.
5. **الرجوع:** إذا لم يكن هناك جيران غير مزارين، نخرج من الرأس الحالي ونعود إلى الرأس الذي سبق زيارته.
6. **التكرار:** نكرر الخطوات 3 و 4 و 5 حتى يتم زيارة جميع الرؤوس المتصلة برأس البداية.

**تنفيذ DFS في بايثون:**

Python

def dfs(graph, start):

visited = set() # مجموعة لتخزين الرؤوس التي تم زيارتها

def dfs\_helper(node):

visited.add(node)

print(node, end=' ')

for neighbor in graph[node]:

if neighbor not in visited:

dfs\_helper(neighbor)

dfs\_helper(start)

# مثال على استخدام الدالة:

graph = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

dfs(graph, 'A') # ستطبع: A B D E F C

يُرجى استخدام الرمز البرمجي [بحذر](/faq#coding).

**شرح الكود:**

* **dfs(graph, start):** هذه هي الدالة الرئيسية التي تأخذ الرسم البياني ونقطة البداية كمدخلات.
* **visited:** مجموعة لتتبع الرؤوس التي تم زيارتها.
* **dfs\_helper(node):** هذه هي الدالة المساعدة التي تقوم بالبحث الفعلي:
  + تضيف الرأس الحالي إلى مجموعة الزيارات.
  + تطبع الرأس الحالي.
  + تتكرر على جميع جيران الرأس الحالي، إذا لم يتم زيارتهم من قبل، تستدعي الدالة نفسها بشكل تكراري.

**ملاحظات:**

* **تمثيل الرسم البياني:** في هذا المثال، يتم تمثيل الرسم البياني باستخدام قاموس حيث المفتاح هو الرأس والقيمة هي قائمة بجيرانه.
* **تعديل الدالة:** يمكن تعديل هذه الدالة لتنفيذ مهام أخرى مثل إيجاد مسار أو الكشف عن دورات.
* **تعقيد زمني:** في أسوأ الحالات، يكون تعقيد الوقت الزمني لـ DFS هو O(V+E)، حيث V هو عدد الرؤوس وE هو عدد الحواف.

**تطبيقات DFS:**

* **حل متاهات:** يمكن تمثيل المتاهة كرسم بياني، ويتم استخدام DFS للعثور على طريق للخروج.
* **تحليل شبكات التواصل الاجتماعي:** يمكن استخدام DFS لتحليل العلاقات بين المستخدمين في شبكة اجتماعية.
* **كشف عن الفيروسات:** يمكن استخدام DFS للكشف عن انتشار الفيروس في شبكة الكمبيوتر.

## شرح خوارزمية البحث العرضي الأول (BFS) وكودها في بايثون

### ما هي خوارزمية البحث العرضي BFS؟

خوارزمية البحث العرضي (Breadth-First Search) هي إحدى الخوارزميات الأساسية في علوم الحاسوب، تستخدم لاستكشاف جميع الرؤوس والروابط في بنية بيانات شجرية أو بيانية. على عكس DFS، تقوم BFS بزيارة جميع الجيران المباشرين لرأس ما قبل الانتقال إلى الجيل التالي من الجيران.

### متى نستخدم BFS؟

* **البحث عن أقصر مسار:** لإيجاد أقصر مسار بين نقطتين في رسم بياني غير موجه وغير مرجح.
* **الكشف عن مكونات متصلة:** لتقسيم الرسم البياني إلى مكونات متصلة.
* **حل المسائل المتعلقة بالبحث:** مثل مشكلة المتاهة.

### كيف تعمل BFS؟

1. **البدء برأس:** نختار رأسًا بدءًا للبحث ونضعه في قائمة الانتظار.
2. **إزالة رأس من القائمة:** نزيل الرأس الأول من قائمة الانتظار.
3. **الزيارة:** نضع علامة على الرأس بأنه تمت زيارته.
4. **إضافة الجيران:** نضيف جميع جيران الرأس الذي تم إزالته إلى قائمة الانتظار إذا لم يتم زيارتها من قبل.
5. **التكرار:** نكرر الخطوات من 2 إلى 4 حتى تصبح قائمة الانتظار فارغة.

### تنفيذ BFS في بايثون:

Python

from collections import deque

def bfs(graph, start):

visited = set() # مجموعة لتخزين الرؤوس التي تم زيارتها

queue = deque([start]) # قائمة الانتظار

while queue:

node = queue.popleft()

if node not in visited:

visited.add(node)

print(node, end=' ')

for neighbor in graph[node]:

queue.append(neighbor)

# مثال على استخدام الدالة:

graph = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

}

bfs(graph, 'A') # ستطبع: A B C D E F

يُرجى استخدام الرمز البرمجي [بحذر](/faq#coding).

### شرح الكود:

* **deque:** تستخدم هنا لتمثيل قائمة الانتظار (FIFO).
* **visited:** مجموعة لتتبع الرؤوس التي تم زيارتها.
* **queue:** قائمة الانتظار التي تحتوي على الرؤوس التي يجب زيارتها.
* **الحلقة while:** تستمر حتى تصبح قائمة الانتظار فارغة.
* **node = queue.popleft():** إزالة الرأس الأول من القائمة.
* **visited.add(node):** إضافة الرأس إلى مجموعة الزيارات.
* **for neighbor in graph[node]::** تكرار على جميع جيران الرأس الحالي.
* **queue.append(neighbor):** إضافة الجيران إلى قائمة الانتظار إذا لم يتم زيارتها من قبل.

### ملاحظات:

* **تمثيل الرسم البياني:** نفس التمثيل المستخدم في DFS.
* **تعديل الدالة:** يمكن تعديل هذه الدالة لإيجاد أقصر مسار أو حل مسائل أخرى.
* **تعقيد زمني:** في أسوأ الحالات، يكون تعقيد الوقت الزمني لـ BFS هو O(V+E)، حيث V هو عدد الرؤوس وE هو عدد الحواف.

**مقارنة بين BFS و DFS:**

|  |  |  |
| --- | --- | --- |
| **الخاصية** | **BFS** | **DFS** |
| استكشاف | عرضي (مستوى بعد مستوى) | عمقي (أقصى عمق ثم العودة) |
| بنية البيانات | قائمة انتظار | مكدس |
| إيجاد أقصر مسار | فعال في الرسوم البيانية غير الموجهة وغير المرجحة | قد لا يكون فعالًا |
| الكشف عن الدورات | قد لا يكون فعالًا | فعال في الكشف عن دورات |

التصدير إلى "جداول بيانات Google"

**تطبيقات BFS:**

* **البحث في شبكات الإنترنت:** للعثور على صفحات الويب المتصلة بصفحة معينة.
* **تحليل شبكات التواصل الاجتماعي:** للعثور على الأصدقاء المشتركين بين المستخدمين.
* **حل مشاكل المتاهات:** للعثور على أقصر طريق للخروج من المتاهة.

شرح خوارزمية Ga في لغة بايثون

## الخوارزميات الجينية (Genetic Algorithms) في بايثون: شرح مبسط وتطبيق عملي

### ما هي الخوارزميات الجينية؟

الخوارزميات الجينية هي تقنية تحسين مستوحاة من عملية التطور البيولوجي. تستخدم لمحاكاة السلوك التطوري لايجاد حلول لمشاكل التحسين. تعمل هذه الخوارزميات عن طريق توليد مجموعة أولية من الحلول المحتملة (الأفراد)، ثم تطور هذه الحلول عبر أجيال متتالية باستخدام عمليات تشبه العمليات البيولوجية مثل التزاوج والطفرات.

### مكونات الخوارزمية الجينية الأساسية:

* **الافراد (Individuals):** تمثل كل فرد حلاً محتملاً للمشكلة.
* **الجينوم (Genome):** يمثل تركيب الفرد، ويتم ترميزه عادة كسلسلة من الأرقام أو الأحرف.
* **الجيل (Generation):** مجموعة من الأفراد في مرحلة معينة من التطور.
* **الدالة التقييمية (Fitness Function):** تحدد مدى جودة كل فرد في حل المشكلة.
* **التزاوج (Crossover):** عملية تبادل أجزاء من جينوم فردين لإنتاج أفراد جدد.
* **الطفرة (Mutation):** عملية تغيير عشوائي في الجينوم لإدخال تنوع جيني.
* **الانتقاء (Selection):** عملية اختيار الأفراد الأفضل للانتقال إلى الجيل التالي.

### خطوات الخوارزمية الجينية:

1. **التوليد الأولي:** يتم إنشاء مجموعة عشوائية من الأفراد.
2. **التقييم:** يتم حساب قيمة الدالة التقييمية لكل فرد.
3. **الانتقاء:** يتم اختيار الأفراد الأفضل للتكاثر بناءً على قيم التقييم.
4. **التزاوج:** يتم تزاوج الأفراد المختارة لإنتاج أفراد جديدة.
5. **الطفرة:** يتم تطبيق الطفرات على بعض الأفراد بشكل عشوائي.
6. **الجيل الجديد:** يتم استبدال الجيل القديم بالجيل الجديد الناتج عن التزاوج والطفرات.
7. **التكرار:** تتكرر الخطوات من 2 إلى 6 حتى الوصول إلى عدد معين من الأجيال أو حتى يتم تحقيق معيار توقف معين.

### مثال على تطبيق الخوارزمية الجينية في بايثون:

Python

import random

def fitness\_function(individual):

# هنا يتم تعريف دالة التقييم التي تحسب جودة الفرد

def crossover(parent1, parent2):

# هنا يتم تنفيذ عملية التزاوج بين فردين

def mutation(individual):

# هنا يتم تنفيذ عملية الطفرة على فرد

# توليد الجيل الأولي

population = generate\_initial\_population()

# الحلقة الرئيسية للخوارزمية

for generation in range(num\_generations):

# التقييم

fitness\_scores = [fitness\_function(individual) for individual in population]

# الانتقاء

parents = select\_parents(population, fitness\_scores)

# التزاوج والطفرات

offspring = []

for \_ in range(len(population)):

parent1, parent2 = random.sample(parents, 2)

child = crossover(parent1, parent2)

child = mutation(child)

offspring.append(child)

# استبدال الجيل القديم بالجديد

population = offspring

# بعد الانتهاء من جميع الأجيال، يكون أفضل فرد في الجيل الأخير هو الحل الأمثل

best\_individual = max(population, key=fitness\_function)

يُرجى استخدام الرمز البرمجي [بحذر](/faq#coding).

### ملاحظات:

* **تحديد المشكلة:** يجب تحديد المشكلة التي تريد حلها بدقة وتحديد طريقة ترميز الحلول.
* **الدالة التقييمية:** هي قلب الخوارزمية، يجب أن تعكس بدقة جودة الحل.
* **عمليات الوراثة:** اختيار عمليات التزاوج والطفرات المناسبة يعتمد على نوع المشكلة.
* **معايير التوقف:** تحديد متى تتوقف الخوارزمية.
* **معالجة التقارب المحلي:** قد تتوقف الخوارزمية في حل محلي وليس الحل الأمثل العالمي.

### مكتبات بايثون لدعم الخوارزميات الجينية:

* **PyGAD:** مكتبة قوية وسهلة الاستخدام لتنفيذ الخوارزميات الجينية.
* **DEAP:** مكتبة مرنة وقابلة للتوسيع لبناء الخوارزميات التطورية.

**تطبيقات الخوارزميات الجينية:**

* **التحسين:** حل مشاكل التحسين المعقدة في مجالات الهندسة، والاقتصاد، والعلوم.
* **التعلم الآلي:** تصميم شبكات عصبية، وتوليد ميزات، وتحديد المعلمات.
* **الذكاء الاصطناعي:** حل مشاكل البحث والإنقاذ، وتخطيط المسارات، والتعلم التعاوني.

**ملاحظات إضافية:**

* الخوارزميات الجينية هي أدوات قوية لحل مجموعة واسعة من المشاكل، ولكنها تتطلب فهمًا جيدًا للمبادئ الأساسية والتجربة لتطبيقها بشكل فعال.
* هناك العديد من التعديلات والتحسينات التي يمكن إجراؤها على الخوارزمية الأساسية لتحسين أدائها.