شرح تفصيلي لخوارزمية A* والكود الخاص بها

جدول المحتويات

مقدمة سريعة

خوارزمية A* هي خوارزمية بحث عن المسار الأمثل تستخدم في:

- الألعاب (تحريك الشخصيات)
- الملاحة (GPS)
- الروبوتات (تخطيط المسار)
- الذكاء الاصطناعي

الفكرة الأساسية: البحث الذكي عن أقصر مسار من نقطة البداية إلى الهدف

فهم الصيغة الأساسية

الصيغة:

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

شرح كل متغير:

$g(n)$ - التكلفة الفعلية من البداية

- المسافة الفعلية التي قطعناها من نقطة البداية إلى العقدة n
- **مثال:** إذا بدأنا من $(0,0)$ ووصلنا إلى $(2,3)$، فإن $g(n)$ = المسافة الفعلية المقطوعة
- **الحساب:**
  - حركة أفقية أو عمودية = 1 وحدة
  - حركة قطرية = 1.414 وحدة $(\sqrt{2})$

$h(n)$ - التكلفة المتوقعة للهدف

- تخمين ذكي للمسافة المتبقية من العقدة n إلى الهدف
- **لا نعرفها بالفعل** - هذا تخمين
- **الهدف:** توجيه البحث نحو الهدف بسرعة

$f(n)$ - التكلفة الإجمالية المتوقعة

- مجموع التكلفة الفعلية والمتوقعة
- **الاستخدام:** نختار العقدة بأقل $f(n)$ للفحص التالي

مثال عملي:

```
(2 ,2) :نقطة البداية
(12 ,12) :نقطة الهدف
(5 ,5) :نقطة حالية

g(5,5) = 4.242 ((5,5) إلى (2,2) من (المسافة الفعلية
h(5,5) = 14 (باستخدام (12,12) إلى (5,5) من المسافة المتوقعة Manhattan)
f(5,5) = 4.242 + 14 = 18.242
```

شرح فئة Node

الكود:

```
class Node {
  x: number;           // الموضع X في الشبكة
  y: number;           // الموضع Y في الشبكة
  g: number;           // التكلفة من البداية
  h: number;           // التكلفة المتوقعة للهدف
  f: number;           // f(n) = g(n) + h(n)
  parent: Node | null; // العقدة الأب (للتتبع)

  constructor(x: number, y: number, g: number, h: number) {
    this.x = x;
    this.y = y;
    this.g = g;
    this.h = h;
    this.f = g + h;  // حساب f مباشرة
    this.parent = null;
  }
}
```

شرح كل خاصية:

x و y

- تمثل موضع العقدة في الشبكة
- **مثال:** Node(5, 3) تعني العقدة في الموضع (5, 3)

g

- التكلفة الفعلية من البداية إلى هذه العقدة
- **مثال:** إذا بدأنا من (0,0) وهذه العقدة في (3,4)، فإن g = 5

h

- التكلفة المتوقعة من هذه العقدة إلى الهدف
- **مثال:** إذا الهدف في (10,10) وهذه العقدة في (3,4)، فإن h = 13 (Manhattan)

f

- مجموع g و h
- **الاستخدام:** نستخدم f لاختيار العقدة الأفضل للفحص التالي

parent

- تخزين العقدة السابقة (الأب)
- **الفائدة:** عند الوصول للهدف، نتتبع الآباء لإعادة بناء المسار الكامل

مثال على الاستخدام:

```
// إنشاء عقدة جديدة
const node = new Node(5, 3, 4.242, 14);
// الآن:
// node.x = 5
// node.y = 3
// node.g = 4.242
// node.h = 14
// node.f = 18.242
// node.parent = null
```

شرح دالة Heuristic

الكود:

```
function heuristic(
  from: Cell,
  to: Cell,
  type: "manhattan" | "euclidean" = "manhattan"
): number {
  const dx = Math.abs(from.x - to.x);  // الفرق في X
  const dy = Math.abs(from.y - to.y);  // الفرق في Y

  if (type === "manhattan") {
    return dx + dy;  // مسافة مانهاتن
  } else {
    return Math.sqrt(dx * dx + dy * dy);  // المسافة الإقليدية
  }
}
```

شرح المتغيرات:

from و to

- `from` : النقطة الحالية
- `to` : النقطة المقصودة (عادة الهدف)

dx و dy

- `dx` : الفرق المطلق بين X في النقطتين
- `dy` : الفرق المطلق بين Y في النقطتين
- **استخدام abs()**: لأننا نريد المسافة الموجبة

نوعا الدوال الإرشادية:

1. Manhattan Distance (مسافة مانهاتن)

```
h(n) = |x_goal - x_n| + |y_goal - y_n|
```

الفكرة: تخيل أنك تتحرك في شارع مدينة (أفقي أو عمودي فقط)

مثال:

```
من (2 ,2) إلى (12 ,12)
dx = |12 - 2| = 10
dy = |12 - 2| = 10
h = 10 + 10 = 20
```

متى تستخدم؟

- عندما تكون الحركة محدودة بـ 4 اتجاهات (أعلى، أسفل، يمين، يسار)
- أكثر دقة في هذه الحالة

2. Euclidean Distance (المسافة الإقليدية)

```
h(n) = √[(x_goal - x_n)² + (y_goal - y_n)²]
```

الفكرة: المسافة المستقيمة بين نقطتين (كما تقيسها بالمسطرة)

مثال:

```
من (2 ,2) إلى (12 ,12)
dx = |12 - 2| = 10
dy = |12 - 2| = 10
h = √(10² + 10²) = √200 = 14.14
```

متى تستخدم؟

- عندما تكون الحركة حرة في 8 اتجاهات (بما فيها الأقطار)
- أكثر واقعية للحركة الحرة

مثال عملي:

```
const from = { x: 2, y: 2 };
const to = { x: 12, y: 12 };

// استخدام Manhattan
const h_manhattan = heuristic(from, to, "manhattan");
// النتيجة: 20

// استخدام Euclidean
const h_euclidean = heuristic(from, to, "euclidean");
// النتيجة: 14.14
```

شرح خوارزمية A* بالتفصيل

الكود الكامل:

```
function aStar(
  start: Cell,
  goal: Cell,
  walls: Set<string>,
  gridSize: number
): Cell[] {
  // 1. تهيئة المجموعات
  const openSet: Node[] = [];                    // العقد المراد فحصها
  const closedSet = new Set<string>();           // العقد المفحوصة بالفعل
  const nodeMap = new Map<string, Node>();       // خريطة سريعة للعقد

  // 2. إنشاء عقدة البداية
  const startNode = new Node(start.x, start.y, 0, heuristic(start, goal));
  openSet.push(startNode);
  nodeMap.set(`${start.x},${start.y}`, startNode);

  // 3. حلقة البحث الرئيسية
  while (openSet.length > 0) {
    // أ. البحث عن العقدة بأقل f(n)
    let current = openSet[0];
    let currentIndex = 0;
    for (let i = 1; i < openSet.length; i++) {
      if (openSet[i].f < current.f) {
        current = openSet[i];
        currentIndex = i;
      }
    }

    // ب. التحقق من الوصول للهدف
    if (current.x === goal.x && current.y === goal.y) {
      // إعادة بناء المسار
      const path: Cell[] = [];
      let node: Node | null = current;
      while (node) {
        path.unshift({ x: node.x, y: node.y });
        node = node.parent;
      }
      return path;
    }
```

```
    // ج. نقل العقدة من openSet إلى closedSet
    openSet.splice(currentIndex, 1);
    closedSet.add(`${current.x},${current.y}`);

    // د. فحص الجيران (8 اتجاهات)
    const neighbors = [
      { x: 0, y: -1 },   // أعلى
      { x: 1, y: 0 },    // يمين
      { x: 0, y: 1 },    // أسفل
      { x: -1, y: 0 },   // يسار
      { x: 1, y: -1 },   // أعلى يمين
      { x: 1, y: 1 },    // أسفل يمين
      { x: -1, y: 1 },   // أسفل يسار
      { x: -1, y: -1 }   // أعلى يسار
    ];

    for (const neighbor of neighbors) {
      const newX = current.x + neighbor.x;
      const newY = current.y + neighbor.y;

      // التحقق من الحدود
      if (newX < 0 || newX >= gridSize || newY < 0 || newY >= gridSize)
continue;

      // التحقق من الجدران
      if (walls.has(`${newX},${newY}`)) continue;

      // التحقق من closedSet
      if (closedSet.has(`${newX},${newY}`)) continue;

      // حساب التكلفة الجديدة
      const newG = current.g + (Math.abs(neighbor.x) + Math.abs(neighbor.y)
=== 2 ? 1.414 : 1);
      const newH = heuristic({ x: newX, y: newY }, goal);
      const newF = newG + newH;

      // التحقق من وجود مسار أفضل
      const existingNode = nodeMap.get(`${newX},${newY}`);
      if (existingNode && existingNode.g <= newG) continue;

      // إنشاء عقدة جديدة
      const newNode = new Node(newX, newY, newG, newH);
      newNode.parent = current;
      openSet.push(newNode);
      nodeMap.set(`${newX},${newY}`, newNode);
```

```
    }
  }

  return []; // لم يتم العثور على مسار
}
```

شرح كل جزء:

الخطوة 1: التهيئة

```
const openSet: Node[] = [];
const closedSet = new Set<string>();
const nodeMap = new Map<string, Node>();
```

الشرح:

- `openSet` : قائمة العقد التي لم نفحصها بعد
- `closedSet` : مجموعة العقد التي فحصناها بالفعل
- `nodeMap` : خريطة سريعة للوصول الفوري للعقد

لماذا nodeMap؟

- البحث في Array يأخذ $O(n)$
- البحث في Map يأخذ $O(1)$

الخطوة 2: إنشاء عقدة البداية

```
const startNode = new Node(start.x, start.y, 0, heuristic(start, goal));
openSet.push(startNode);
nodeMap.set(`${start.x},${start.y}`, startNode);
```

الشرح:

- `g = 0` : لم نقطع أي مسافة بعد
- `h = heuristic(start, goal)` : المسافة المتوقعة من البداية للهدف

- نضيفها إلى openSet للبدء

الخطوة 3: حلقة البحث

```
while (openSet.length > 0) {
```

الشرح:

- تستمر الحلقة طالما هناك عقد لم نفحصها
- إذا انتهت openSet ولم نجد الهدف = لا يوجد مسار

الخطوة 3.أ: البحث عن أفضل عقدة

```
let current = openSet[0];
let currentIndex = 0;
for (let i = 1; i < openSet.length; i++) {
  if (openSet[i].f < current.f) {
    current = openSet[i];
    currentIndex = i;
  }
}
```

الشرح:

- نبحث عن العقدة بأقل $f(n)$
- هذه هي العقدة الأكثر واعدة
- لماذا أقل f؟ لأنها أقرب للهدف

مثال:

```
openSet = [
  Node(5,5, f=18),
  Node(3,3, f=15),  ← الأقل
  Node(7,7, f=20)
]
current = Node(3,3, f=15)
```

الخطوة 3.ب: التحقق من الهدف

```
if (current.x === goal.x && current.y === goal.y) {
  // إعادة بناء المسار
  const path: Cell[] = [];
  let node: Node | null = current;
  while (node) {
    path.unshift({ x: node.x, y: node.y });
    node = node.parent;
  }
  return path;
}
```

الشرح:

- إذا وصلنا للهدف، نبني المسار بالعودة للآباء
- `path.unshift()` : إضافة في البداية (لأننا نعود للخلف)
- `node = node.parent` : الانتقال للعقدة الأب

مثال على إعادة البناء:

```
current = Node(12,12, parent=Node(11,11, parent=Node(10,10,
parent=Node(2,2, parent=null))))

المسار المبني:
[
  {x:2, y:2},
  {x:10, y:10},
  {x:11, y:11},
  {x:12, y:12}
]
```

الخطوة 3.ج: نقل العقدة للمفحوصة

```
openSet.splice(currentIndex, 1);
closedSet.add(`${current.x},${current.y}`);
```

الشرح:

- `splice()` : إزالة العقدة من openSet
- `add()` : إضافة العقدة إلى closedSet
- لن نفحص هذه العقدة مرة أخرى

الخطوة 3.د: فحص الجيران

```
const neighbors = [
  { x: 0, y: -1 },   // أعلى
  { x: 1, y: 0 },    // يمين
  { x: 0, y: 1 },    // أسفل
  { x: -1, y: 0 },   // يسار
  { x: 1, y: -1 },   // أعلى يمين
  { x: 1, y: 1 },    // أسفل يمين
  { x: -1, y: 1 },   // أسفل يسار
  { x: -1, y: -1 }   // أعلى يسار
];
```

الشرح:

- 8 اتجاهات حركة ممكنة
- كل جار هو إزاحة من الموضع الحالي

مثال:

```
إذا current = (5, 5)
الجيران:
- (5, 4) أعلى
- (6, 5) يمين
- (5, 6) أسفل
- (4, 5) يسار
- (6, 4) أعلى يمين
- (6, 6) أسفل يمين
- (4, 6) أسفل يسار
- (4, 4) أعلى يسار
```

فحص صحة الجار

```
for (const neighbor of neighbors) {
  const newX = current.x + neighbor.x;
  const newY = current.y + neighbor.y;

  // التحقق من الحدود
  if (newX < 0 || newX >= gridSize || newY < 0 || newY >= gridSize)
continue;

  // التحقق من الجدران
  if (walls.has(`${newX},${newY}`)) continue;

  // التحقق من closedSet
  if (closedSet.has(`${newX},${newY}`)) continue;
```

الشرح:

- `continue` : تخطي هذا الجار والانتقال للتالي
- **الحدود:** هل الموضع داخل الشبكة؟
- **الجدران:** هل هناك جدار في هذا الموضع؟
- **closedSet:** هل فحصنا هذا الموضع بالفعل؟

حساب التكلفة الجديدة

```
const newG = current.g + (Math.abs(neighbor.x) + Math.abs(neighbor.y) === 2
? 1.414 : 1);
const newH = heuristic({ x: newX, y: newY }, goal);
const newF = newG + newH;
```

الشرح:

- `newG` : التكلفة من البداية إلى الجار الجديد
  - إذا كانت حركة قطرية (2 في الإزاحة) = 1.414
  - إذا كانت حركة أفقية/عمودية = 1
- `newH` : المسافة المتوقعة من الجار للهدف
- `newF` : مجموع التكلفة الإجمالية

مثال:

```
current.g = 5
الجار في (6 ,6) من (5 ,5) = حركة قطرية
newG = 5 + 1.414 = 6.414
newH = heuristic((6,6), goal) = 10
newF = 6.414 + 10 = 16.414
```

التحقق من وجود مسار أفضل

```
const existingNode = nodeMap.get(`${newX},${newY}`);
if (existingNode && existingNode.g <= newG) continue;
```

الشرح:

- هل فحصنا هذا الموضع من قبل؟
- إذا كان g السابق أقل أو يساوي الجديد = لا نحدث
- السبب: المسار السابق أفضل أو متساوي

إضافة الجار الجديد

```
const newNode = new Node(newX, newY, newG, newH);
newNode.parent = current;
openSet.push(newNode);
nodeMap.set(`${newX},${newY}`, newNode);
```

الشرح:

- إنشاء عقدة جديدة للجار
- تعيين parent = current (لتتبع المسار)
- إضافة إلى openSet للفحص لاحقاً
- إضافة إلى nodeMap للوصول السريع

خطوة بخطوة مع أمثلة

مثال عملي كامل:

السيناريو:

- البداية: (2, 2)
- الهدف: (4, 4)
- الشبكة: 6×6
- لا توجد جدران

التكرار الأول:

```
1. البداية:
   openSet = [Node(2,2, g=0, h=4, f=4)]
   closedSet = []

2. اختيار أفضل عقدة:
   current = Node(2,2, g=0, h=4, f=4)

3. هل وصلنا للهدف؟
   (2,2) != (4,4) → لا

4. نقل للمفحوصة:
   openSet = []
   closedSet = {"2,2"}

5. فحص الجيران:
   (2,1) الجار: newG=1, newH=5, newF=6 → إضافة
   (3,2) الجار: newG=1, newH=3, newF=4 → إضافة
   (2,3) الجار: newG=1, newH=3, newF=4 → إضافة
   (3,1) الجار: newG=1.414, newH=4.24, newF=5.65 → إضافة
   ... وهكذا

6. النتيجة:
   openSet = [
     Node(3,2, g=1, h=3, f=4),
     Node(2,3, g=1, h=3, f=4),
     Node(2,1, g=1, h=5, f=6),
     Node(3,1, g=1.414, h=4.24, f=5.65),
     ...
   ]
```

التكرار الثاني:

```
1. اختيار أفضل عقدة:
   current = Node(3,2, g=1, h=3, f=4)  ← الأقل f

2. هل وصلنا للهدف؟
   (3,2) != (4,4) → لا

3. نقل للمفحوصة:
   closedSet = {"2,2", "3,2"}

4. فحص الجيران:
   (4,3) الجار: newG=2, newH=1.41, newF=3.41 → إضافة
   ... وهكذا
```

التكرار النهائي:

```
1. اختيار أفضل عقدة:
   current = Node(4,4, g=2.828, h=0, f=2.828)

2. هل وصلنا للهدف؟
   (4,4) == (4,4) → نعم!

3. إعادة بناء المسار:
   node = Node(4,4, parent=Node(3,3, parent=Node(2,2, parent=null)))

   المسار:
   [
     {x:2, y:2},
     {x:3, y:3},
     {x:4, y:4}
   ]

4. إرجاع المسار
```

أسئلة المقابلة المتوقعة

السؤال 1: ما هي خوارزمية A*؟

الإجابة: خوارزمية A* هي خوارزمية بحث عن المسار الأمثل تجمع بين:

- Dijkstra: للبحث الدقيق عن أقصر مسار
- Greedy Best-First: للبحث الموجه نحو الهدف

تستخدم الصيغة $f(n) = g(n) + h(n)$ حيث:

- $g(n)$: التكلفة الفعلية من البداية
- $h(n)$: التكلفة المتوقعة للهدف

السؤال 2: ما الفرق بين g و h؟

الإجابة:

- **$g(n)$**: التكلفة الفعلية المقطوعة من البداية إلى العقدة الحالية (معروفة)
- **$h(n)$**: تخمين للمسافة المتبقية من العقدة الحالية للهدف (غير معروفة)

مثال:

```
من (0,0) إلى (10,10)، نحن في (3,4)
g(3,4) = 5 (المسافة الفعلية)
h(3,4) = 13 (Manhattan: |10-3| + |10-4|)
f(3,4) = 18
```

السؤال 3: لماذا نستخدم openSet و closedSet؟

الإجابة:

- **openSet**: العقد المرشحة للفحص (لم نفحصها بعد)
- **closedSet**: العقد التي فحصناها بالفعل

الفائدة: تجنب إعادة فحص نفس العقدة مرتين

السؤال 4: كيف نختار العقدة التالية؟

الإجابة: نختار العقدة بأقل $f(n)$ من openSet

- السبب: هذه العقدة الأكثر واعدة للوصول للهدف
- الخوارزمية توازن بين المسافة المقطوعة والمسافة المتبقية

السؤال 5: ما الفرق بين Manhattan و Euclidean؟

الإجابة:

- **Manhattan:** $h = |x\_goal - x| + |y\_goal - y|$
  - للحركة بـ 4 اتجاهات فقط
  - أكثر دقة في هذه الحالة
- **Euclidean:** $h = \sqrt{[(x\_goal - x)^2 + (y\_goal - y)^2]}$
  - للحركة بـ 8 اتجاهات
  - أكثر واقعية للحركة الحرة

السؤال 6: كيف نعيد بناء المسار؟

الإجابة: عندما نصل للهدف، نتتبع الآباء للخلف:

```
current = goal
path = []
while (current != null) {
  path.add(current)
  current = current.parent
}
```

السؤال 7: ما التعقيد الزمني؟

الإجابة: $O(b^d)$ حيث:

- b: عامل التفرع (عدد الجيران المحتملين)
- d: عمق الحل (عدد الخطوات)

في لعبتنا: b=8 (8 اتجاهات)

السؤال 8: هل الخوارزمية تجد أقصر مسار؟

الإجابة: نعم، إذا كانت الدالة الإرشادية **admissible** (لا تبالغ في التقدير)

- Manhattan و Euclidean كلاهما admissible
- لذلك الخوارزمية تجد أقصر مسار

السؤال 9: ماذا يحدث إذا لم يوجد مسار؟

الإجابة: إذا انتهت openSet ولم نجد الهدف، نرجع مصفوفة فارغة []

- هذا يعني لا يوجد مسار من البداية للهدف

السؤال 10: كيف تتعامل مع الجدران؟

الإجابة: قبل إضافة جار إلى openSet، نتحقق:

```
if (walls.has(`${newX},${newY}`)) continue;
```

إذا كان هناك جدار، نتخطاه ولا نضيفه

ملخص سريع

المفهوم	الشرح
f(n)	التكلفة الإجمالية = $g(n) + h(n)$
g(n)	التكلفة الفعلية من البداية
h(n)	التكلفة المتوقعة للهدف
openSet	العقد المرشحة للفحص
closedSet	العقد المفحوصة
parent	العقدة السابقة (لإعادة بناء المسار)
Manhattan	$= h$
Euclidean	$h = \sqrt{[(x\_goal - x)^2 + (y\_goal - y)^2]}$

نصائح للمقابلة

1. **فهم الصيغة أولاً**: $f(n) = g(n) + h(n)$
2. **شرح openSet و closedSet**: أساسي جداً
3. **أعطِ أمثلة عملية**: أفضل من الشرح النظري
4. **اشرح الفرق بين g و h**: سؤال شائع جداً
5. **تحدث عن التعقيد الزمني**: يظهر فهمك العميق
6. **اشرح لماذا أقل f؟**: هذا يظهر فهمك للخوارزمية

تم الإنشاء: ديسمبر 2025 **الهدف**: تحضير شامل للمقابلة العملية