# Search Algorithms

**CE2103 - Algorithms and Data Structures II**

## Disclaimer / Descargo de Responsabilidad

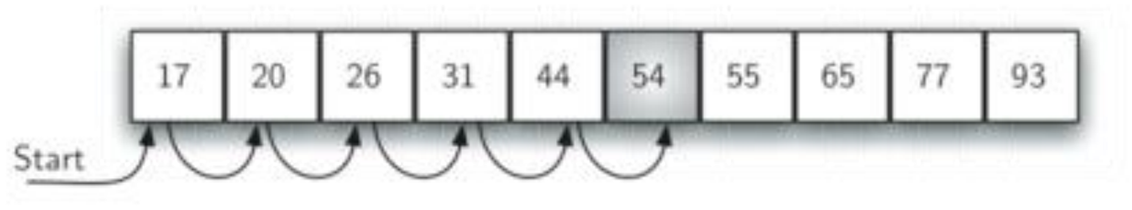
## Levels of Complexity

| | | |
|---|---|---|
| 1 | Constant | Common Problems |
| log N | Logarithmic | |
| N | Linear | |
| N log N | Linear Logarithmic | |
| N^2 | quadratic | |
| N^3 | qubic | |
| 2^N | Exponential | Hard Problems |
| N! | Factorial | |

## Sequential Search

Complexity:
O(n)

- Is the simplest search algorithm.
- Used to search for an element in a list or vector.
- In the worst case, as many comparison as elements in the array.



## Sequential Search (Code)

**Complexity:**
**O(n)**

➔ If the array is **not sorted**

```
int i = 0;
for(; i < N; i++) {
   if(array[i] == number)
      break;
}
if(i == N)
   i = -1;
```

## Sequential Search (Code)

**Complexity:**
**O(n)**

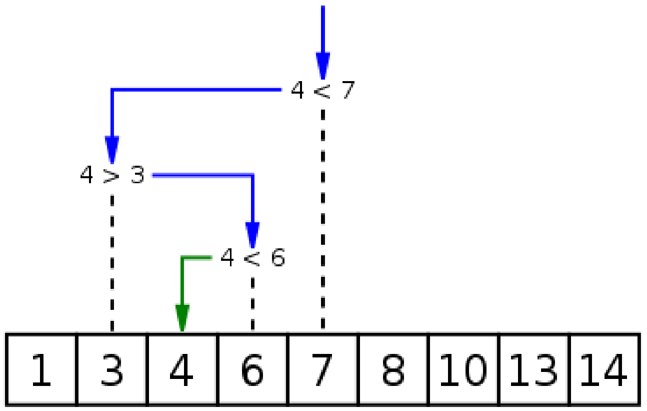
➔ If the array is **sorted**

```
int i = 0;
for(; i < N; i++){
   if(array[i] == number)
      break;
   if(array[i] > number) {
      i = -1;
      break;
   }
}
if(i == N)
   i = -1;
```

## Binary Search

**Complexity:**
**O(log (n))**

- ➔ Finds a number in a **sorted** array.
- ➔ Compares the input element with the **middle** of the array.
- ➔ If the input element matches, return the index to that element.
- ➔ Otherwise, runs the algorithm with the right or left **subarray**, depending if the input key is greater or less than the middle element.
- ➔ Array must be sorted.



## Binary Search (Code)

**Complexity:**
**O(log (n))**

```
int binarySearch(int pArray[], int pKey, int pIndexMin, int pIndexMax)
{
   while (pIndexMax >= pIndexMin)
   {
     int middle = (int)((pIndexMax + pIndexMin) / 2);

     if (pArray[middle] < pkey)
        pIndexMin = middle + 1;
     else if (pArray[middle] > pKey)
        pIndexMax = middle - 1;
     else
        return middle;
   }
   return -1;
}
```

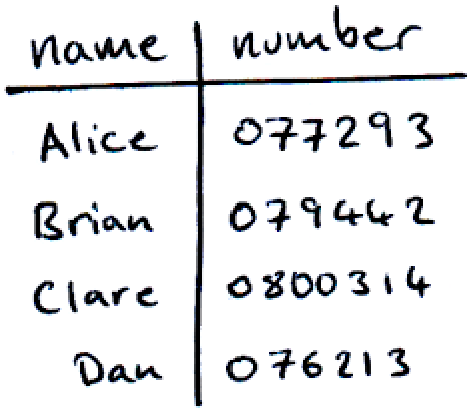
## Interpolation Search

**Complexity: O(log log (n))**

➔ Modification of Binary Search.

➔ In each step tries to calculate where the number might be.

➔ Based on the idea of looking for a person in the phonebook. If you're looking for Bob, you know it should be at the beginning.



| name | number |
|---|---|
| Alice | 077293 |
| Brian | 079442 |
| Clare | 0800314 |
| Dan | 076213 |

## Interpolation Search

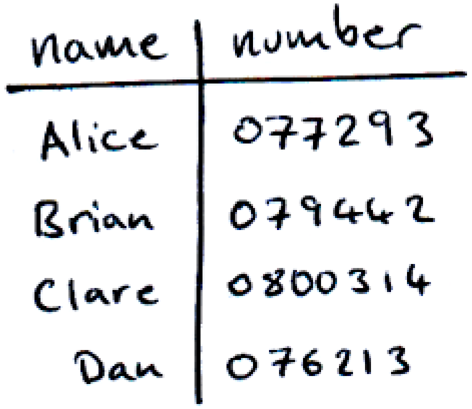
**Complexity: O(log log (n))**

**If elements are uniformly distributed, otherwise can be O(n)**

➔ Modification of Binary Search.

➔ In each step tries to calculate where the num

➔ Based on the idea of looking for a person in the phonebook. If you’re looking for Bob, you know it should be at the beginning.



| name | number |
|---|---|
| Alice | 077293 |
| Brian | 079442 |
| Clare | 0800314 |
| Dan | 076213 |

# Interpolation Search (Code)

**Complexity: O(log log (n))**

```
middle = low + ((number - array[low]) * (high -
low)) / (array[high] - array[low]);
```

**Rest of the code is equal to Binary Search**

## Interpolation Search (Example)

**Complexity: O(log log (n))**

```
middle = low + ((number - array[low]) * (high - low))
       / (array[high] - array[low]);
```

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 6 | 9 | 11 | 15 | 18 | 20 | 25 | 28 | 39 |

*Let’s search for 28...*

## Interpolation Search (Example)

**Complexity: O(log log (n))**

```
middle = low + ((number - array[low]) * (high - low))
        / (array[high] - array[low]);
```

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 6 | 9 | 11 | 15 | 18 | 20 | 25 | 28 | 39 |

*The middle will be... 0 + ((28-5) * (10-0)) / (39 - 5) = 6,76 = 7*

## Interpolation Search (Example)

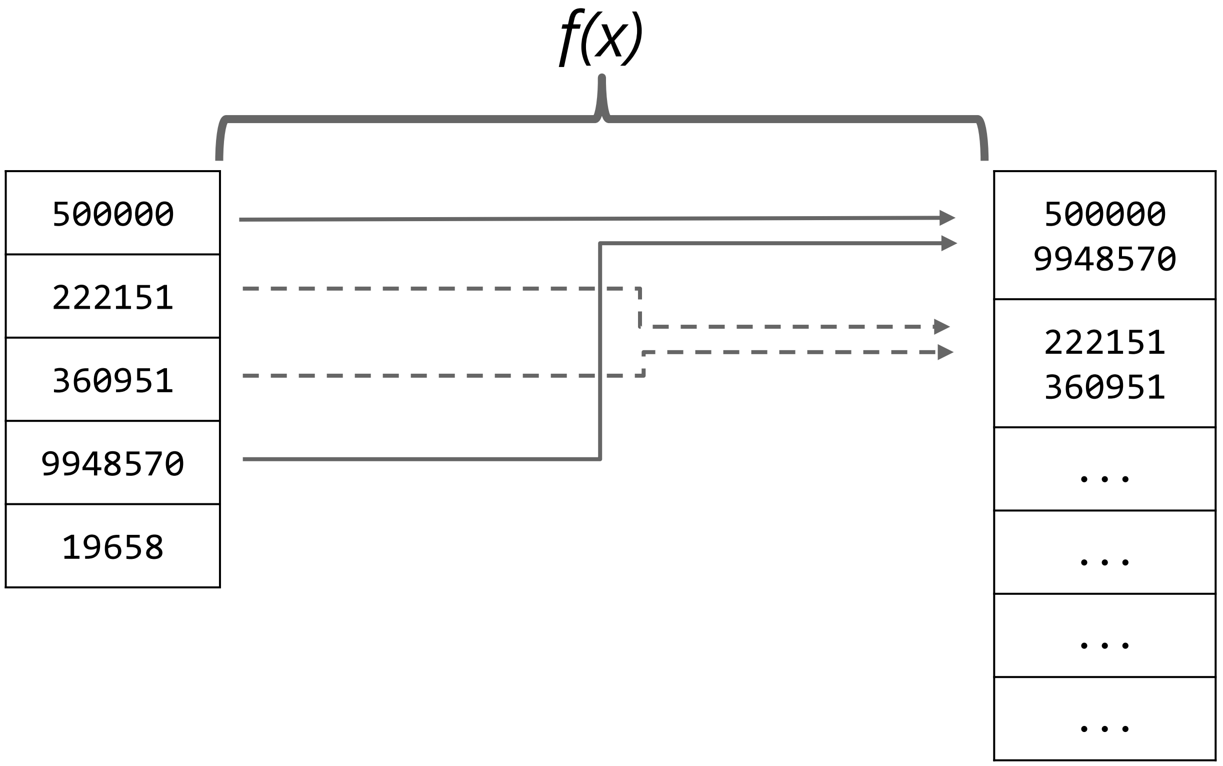
**Complexity: O(log log (n))**

```
middle = low + ((number - array[low]) * (high - low))
        / (array[high] - array[low]);
```

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 6 | 9 | 11 | 15 | 18 | 20 | 25 | 28 | 39 |

*The middle will be... 0 + ((28-5) * (10-0)) / (39 - 5) = 6,76 = 7*

# Hashing



f(x)
500000
222151
360951
9948570
19658
500000
9948570
222151
360951
...
...
...
...

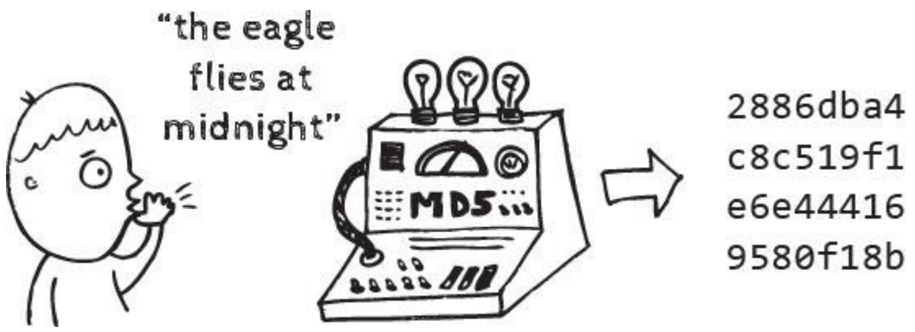
## Hashing

**CHARACTERISTICS**

- ➔ Maps large sets of data to small sets.
- ➔ It's a fast search method.
- ➔ Hash function allows find and assign an index to a key value.
- ➔ It can map several keys to the same index.

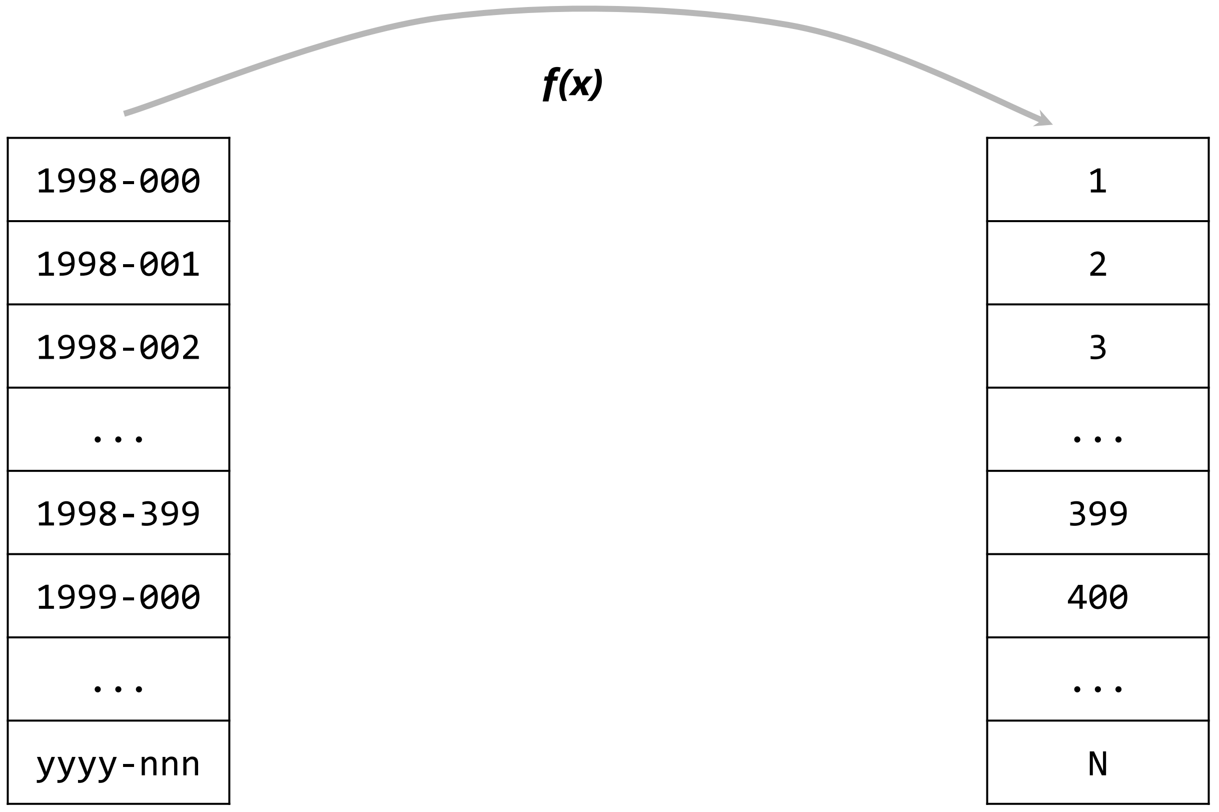
**MORE CHARACTERISTICS**

- ➔ Each slot in the hash table has assigned a set of data. Each slot is called bucket.
- ➔ Transforms keys to indexes.
- ➔ The basic hash function for numbers us the **identity function**. It is not used.



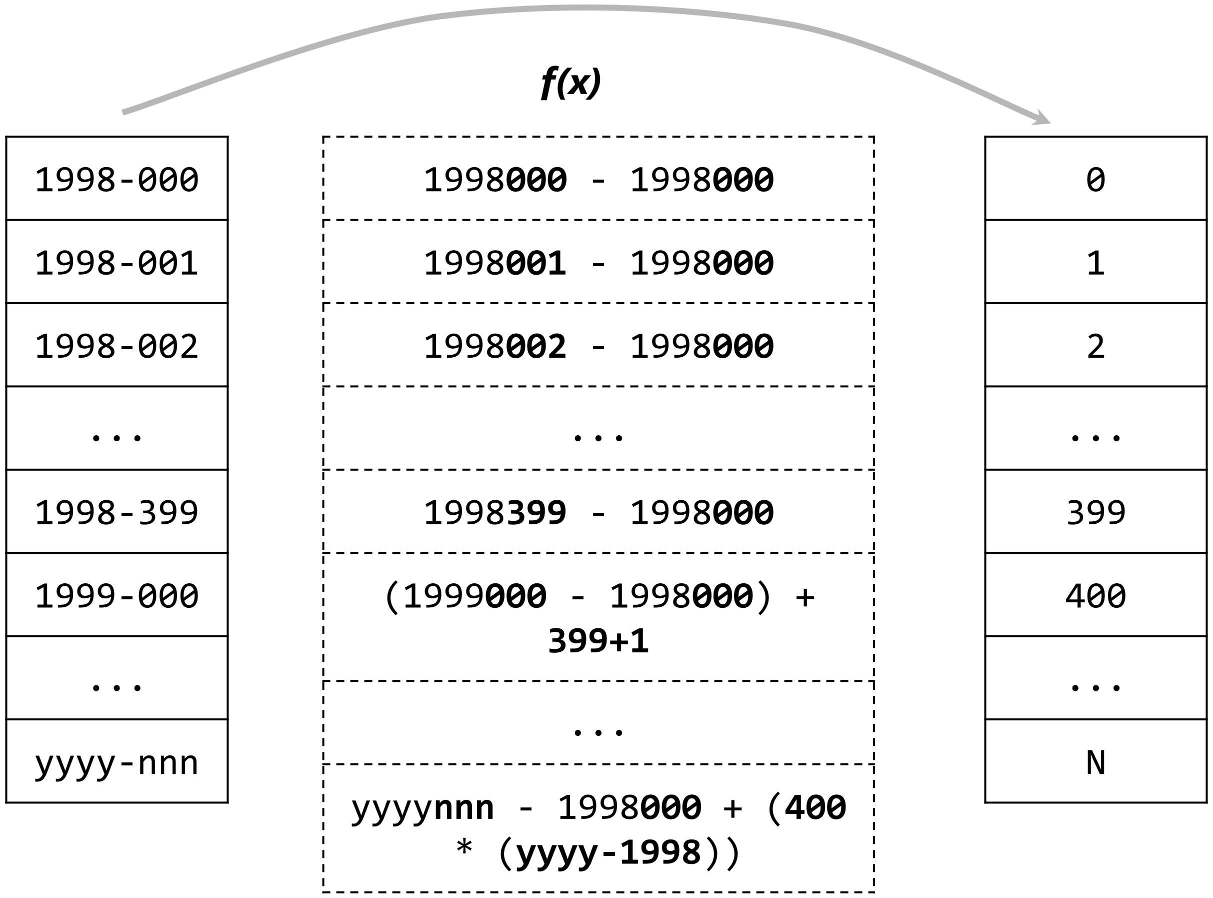
## Hashing (Function)

➔ Successive subtraction



## Hashing (Function)

➔ Successive subtraction



# Hashing (Function)

➔ Modular Arithmetic

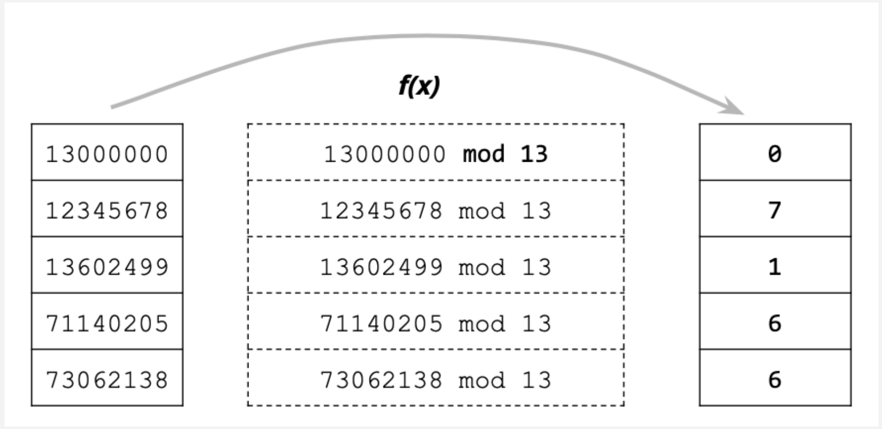
**STRATEGY**

➔ Use a prime number.

➔ Index is the residue of divide the key between a number (module).

➔ The number defines the amount of buckets of the hash table.

**EXAMPLE**



## Hashing (Function)

➔ Mid-Square Method

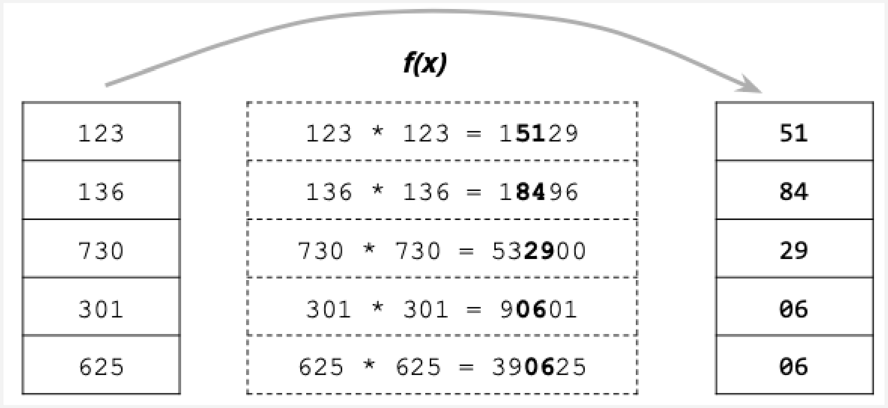
**STRATEGY**

➔ Square the key value.

➔ Takes the middle *r* digits of the result.

➔ It gives a value between 0 and ( 2^r ) -1

**EXAMPLE**



| | | |
|---|---|---|
| 123 | 123 * 123 = 1**51**29 | **51** |
| 136 | 136 * 136 = 1**84**96 | **84** |
| 730 | 730 * 730 = 53**29**00 | **29** |
| 301 | 301 * 301 = 9**06**01 | **06** |
| 625 | 625 * 625 = 39**06**25 | **06** |

## Hashing (Function)

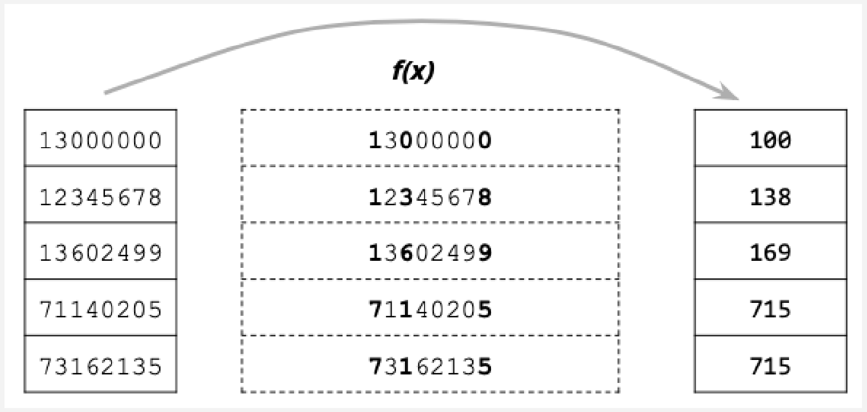
➔ Truncation Method

**STRATEGY**

➔ Ignore part of the key and use the rest as the array index.

➔ You don’t need to get successive numbers.

**EXAMPLE**



| | | |
|---|---|---|
| 13000000 | **1**3**0**0000**0** | 100 |
| 12345678 | **1**2**3**4567**8** | 138 |
| 13602499 | **1**3**6**0249**9** | 169 |
| 71140205 | **7**1**1**4020**5** | 715 |
| 73162135 | **7**3**1**6213**5** | 715 |

## Hashing (Function)

➔ Folding Method

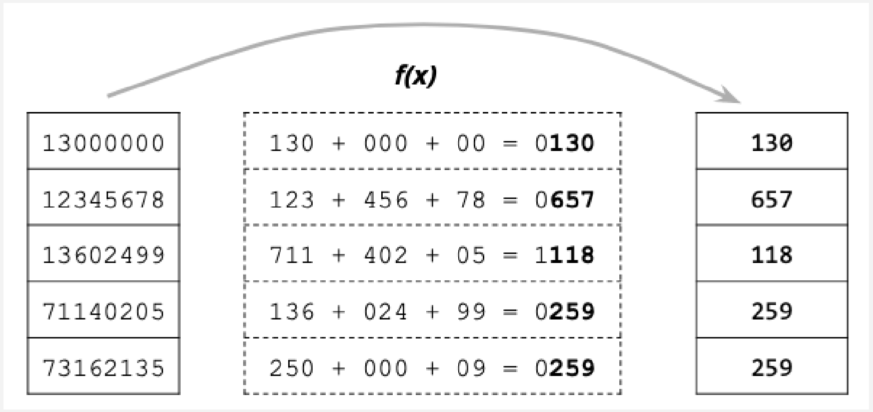
**STRATEGY**

➔ Divide the key in parts.

➔ Combine this parts (might be using operator / * + -).

➔ For example, divide a number of 8 digits in groups of 3 digits and sum this groups.

**EXAMPLE**



| | | |
|---|---|---|
| 13000000 | 130 + 000 + 00 = 0**130** | **130** |
| 12345678 | 123 + 456 + 78 = 0**657** | **657** |
| 13602499 | 711 + 402 + 05 = 1**118** | **118** |
| 71140205 | 136 + 024 + 99 = 0**259** | **259** |
| 73162135 | 250 + 000 + 09 = 0**259** | **259** |

## Hashing (Function)

**What’s the problem of Hash Tables?**

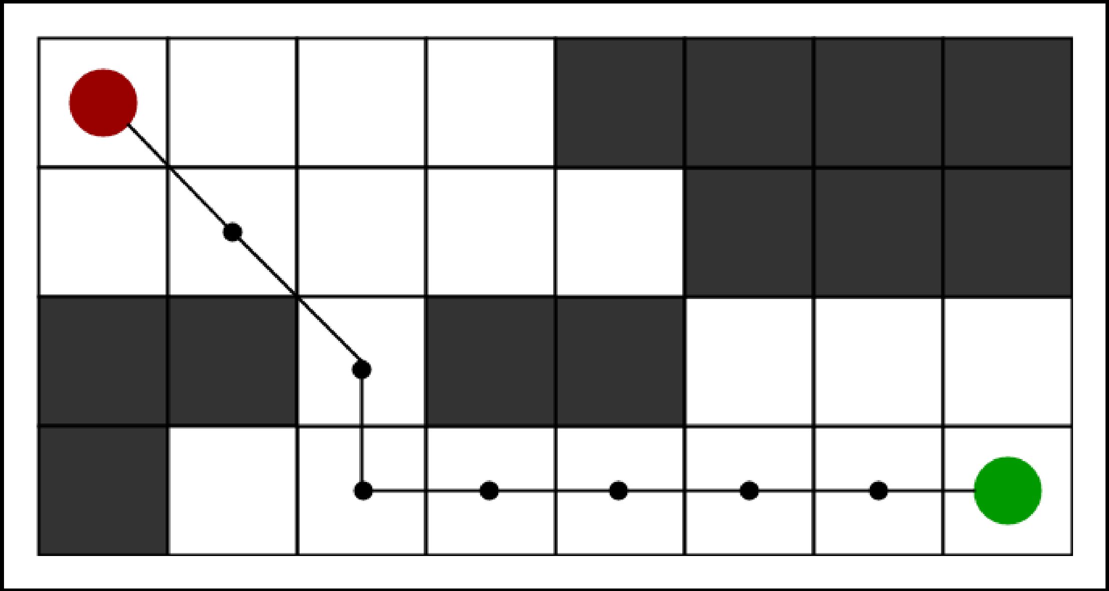
## Hashing (Collisions)

- Two or more keys with the same index.
- Collisions are practically unavoidable. Collisions Treatment in some cases is very expensive.
- Wrong choice of hash function can increase this problem.
- Almost all the hash slots remaining are empty while a few are full and present a lot of collisions.
- Small hash table and too much keys to be sorted.

## Hashing

How to deal with collisions?

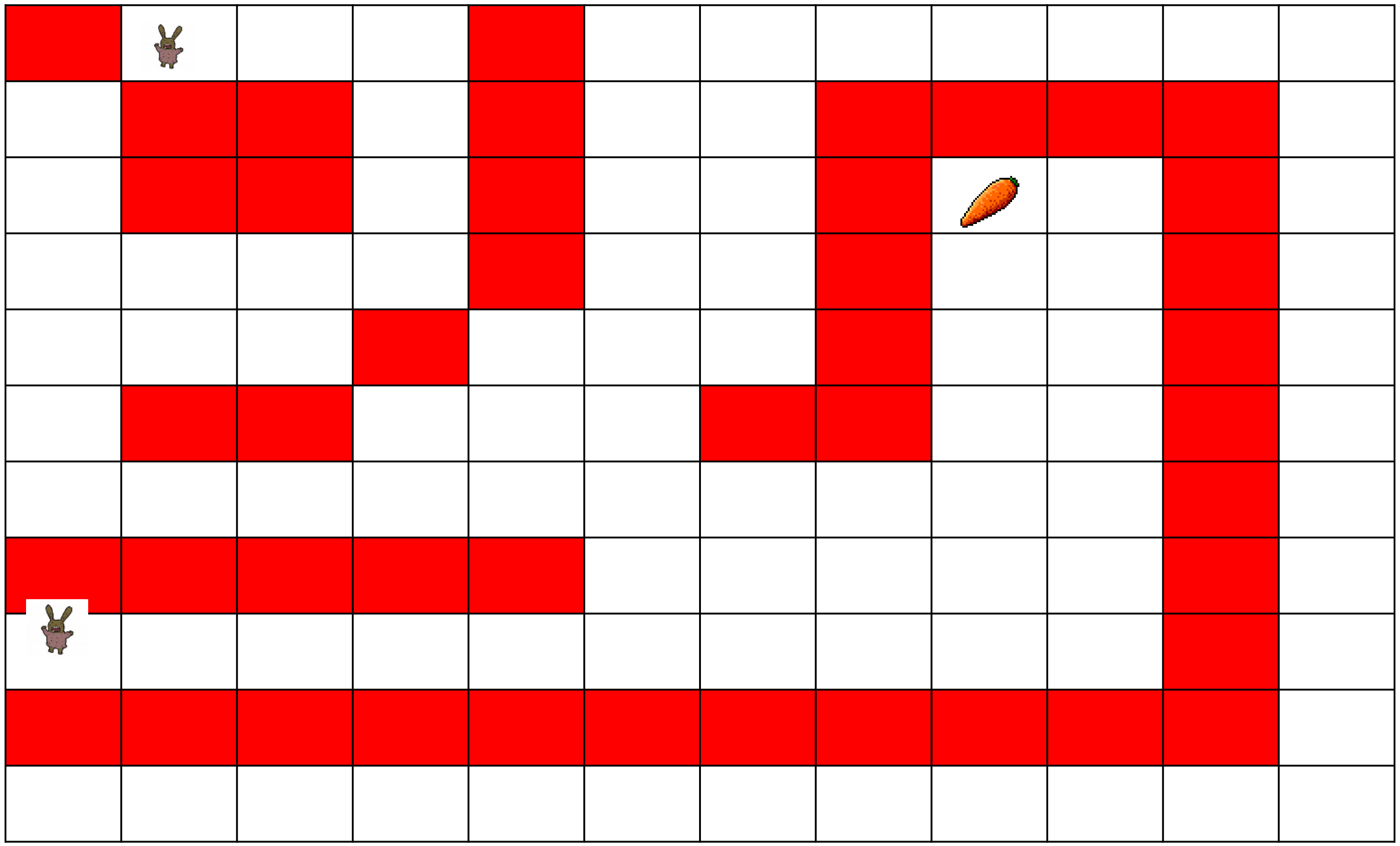
## Pathfinding



## Pathfinding

| CHARACTERISTICS | MORE CHARACTERISTICS |
|---|---|
| ➔ Finds the shortest route between two points.<br>➔ Used to solve mazes.<br>➔ We can abstract its functionality by saying its is an algorithm that explore adjacent nodes and selects the closest. | ➔ It is usually implemented as an algorithm that runs over a matrix.<br>➔ Each matrix cell is a node.<br>➔ These kind of algorithm are used in strategy games. |

# Pathfinding

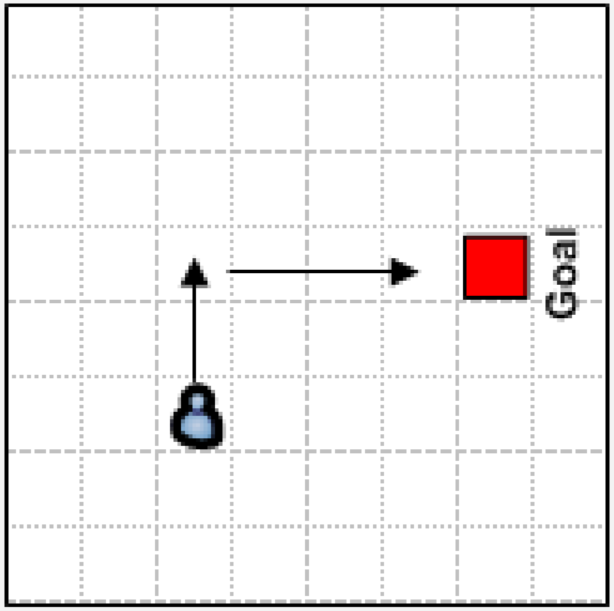


## Pathfinding (Techniques)

- ➔ There are many ways to implement Pathfinding.
- ➔ Depends on the context:
  - ◆ Is the goal moving or stationary?
  - ◆ Are there obstacles?
  - ◆ Is the shortest solution always the best solution?
  - ◆ Is it required to reach a specific destination or just a partial route will do?

## Pathfinding (Techniques)

→ Simplest Form

| STRATEGY | EXAMPLE |
|---|---|
| → **There isn't obstacles**. Just compare X and Y axis. |  |

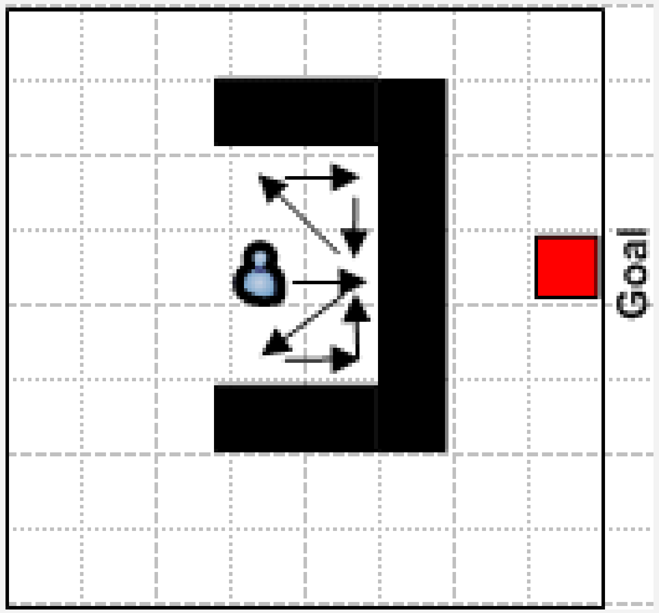
## Pathfinding (Techniques)

- → Random

**STRATEGY**

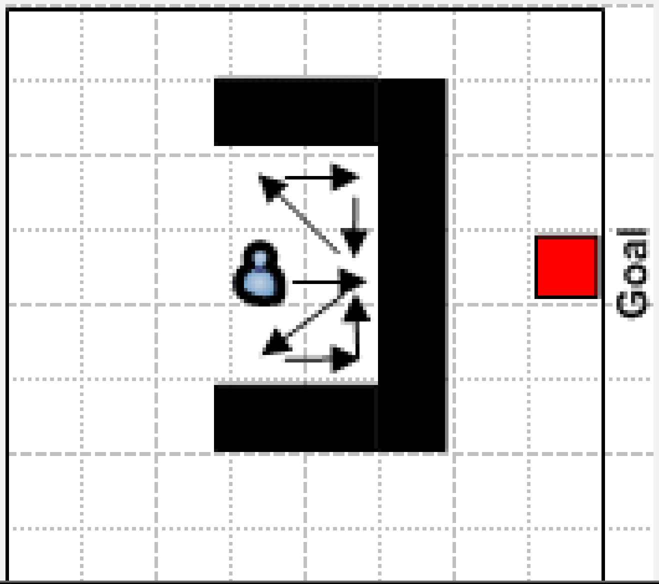
- → Take one step at a time in the direction of the goal.
- → If an obstacle is encountered try to work around it by backstepping a bit **in a random direction** and then try again.

**EXAMPLE**



# Pathfinding (Techniques)

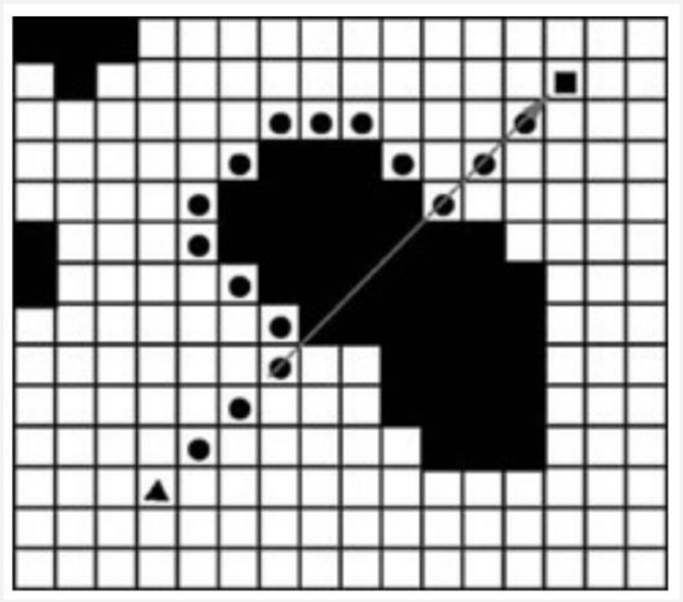
➔ Random

| STRATEGY | EXAMPLE |
|---|---|
| ➔ Take one step at a time in the direction of the goal.<br><br>➔ If an obstacle is encountered **takes a random adjacent cell** and tries again. |  |

- ➔ *Simplest form with obstacles.*
- ➔ *It shouldn't be used. Algorithm will have a "stupid" behaviour.*
- ➔ *Usually, this algorithm does not find the goal.* ***Will get stuck in a multitude of situations.***

## Pathfinding (Techniques)

→ Obstacle Tracing

| STRATEGY | EXAMPLE |
|---|---|
| → Take one step at a time in the direction of the goal.<br><br>→ If an obstacle is encountered start **tracing around the object**.<br><br>→ Once there's no obstacle continue moving in the goal's direction. |  |

## Pathfinding (Techniques)

→ Obstacle Tracing

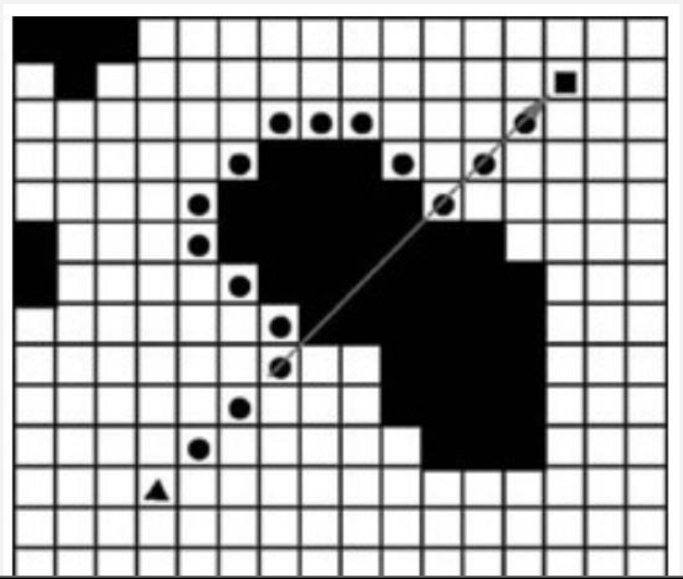
**STRATEGY**

→ Take one step at a time in the direction of the goal.

→ If an obstacle is encountered start **tracing around the object**.

→ Once there's no obstacle continue moving in the goal's

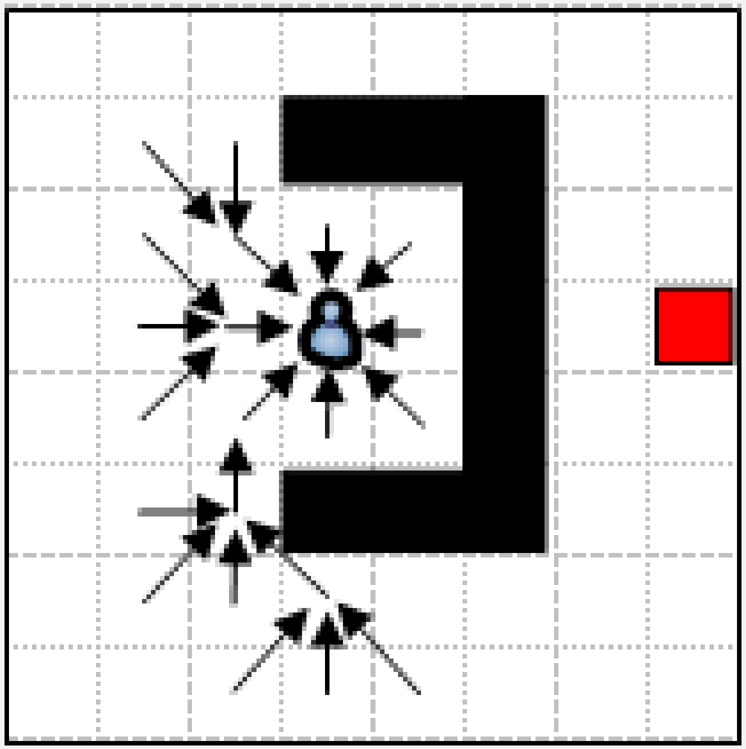
**EXAMPLE**



→ *It can also **get stuck in a multitude of situations**.*

# Pathfinding (Techniques)

## ➔ Breadth-First Search (BFS)

| STRATEGY | EXAMPLE |
|---|---|
| ➔ Traverses a tree structure by fanning out to **explore the nearest neighbors and then their sublevel neighbors**.<br><br>➔ From the initial cell, visit the cells at one jump of distance.<br><br>➔ If you don't find the goal, visit at two cells of distance, if you don't find the goal, visit at three jumps and so on.<br><br>➔ It stops when it finds the goal. |  |

# Pathfinding (Techniques)

➔ Breadth-First Search (BFS)

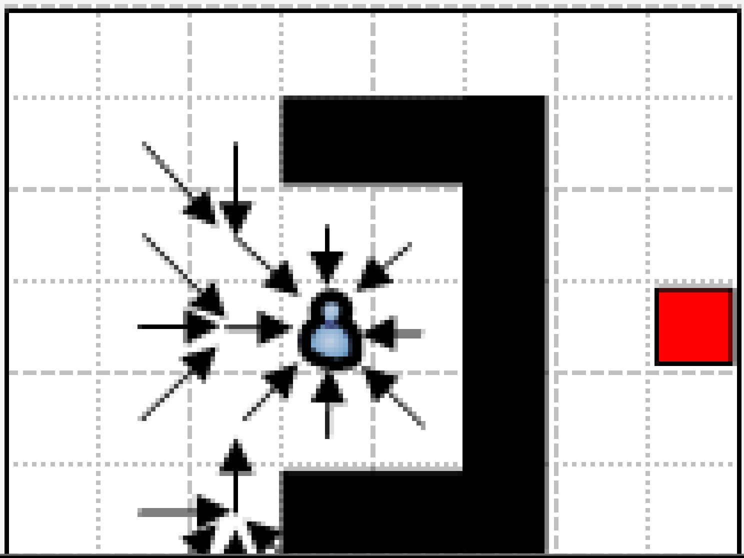
**STRATEGY**

➔ Traverses a tree structure by fanning out to **explore the nearest neighbors and then their sublevel neighbors**.

➔ From the initial cell, visit the cells at one jump of distance.

➔ If you don't find the goal, visit at two cells of distance, if you don't find the

**EXAMPLE**

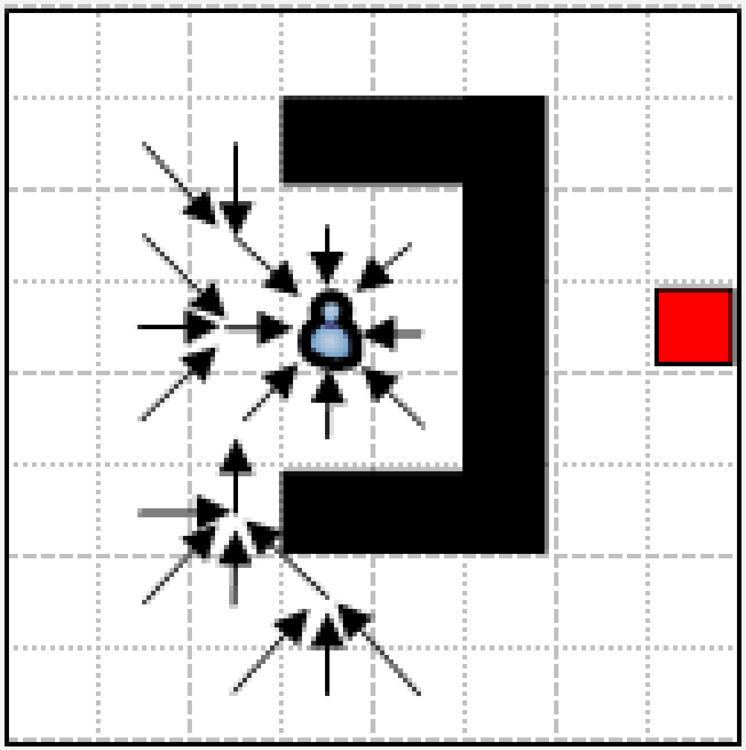


➔ *If there is path to the goal, this algorithm will find it. Maybe no the shortest path.*

➔ ***Inefficient****. If the graph is unweighted it funds the shortest path although not too efficiently.*

# Pathfinding (Techniques)

➔ Depth-First Search (DFS)

| STRATEGY | EXAMPLE |
|---|---|
| ➔ Traverses a tree structure by **exploring as far as possible down each branch** before backtracking. |  |

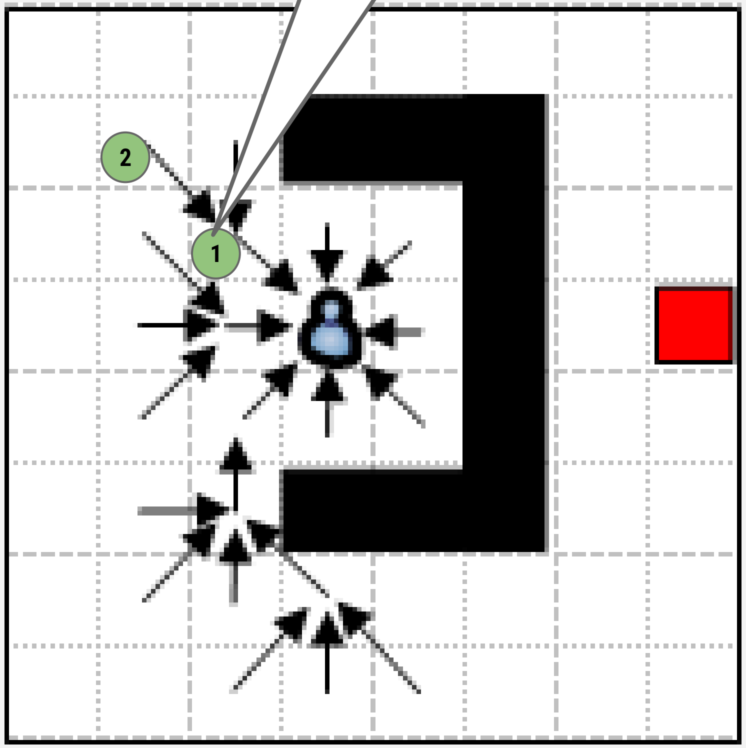
# Pathfinding (Techniques)

→ Depth-First Search (DFS)

In step 1, select this cell and pause exploring the other in the same level.

| STRATEGY | EXAMPLE |
|---|---|

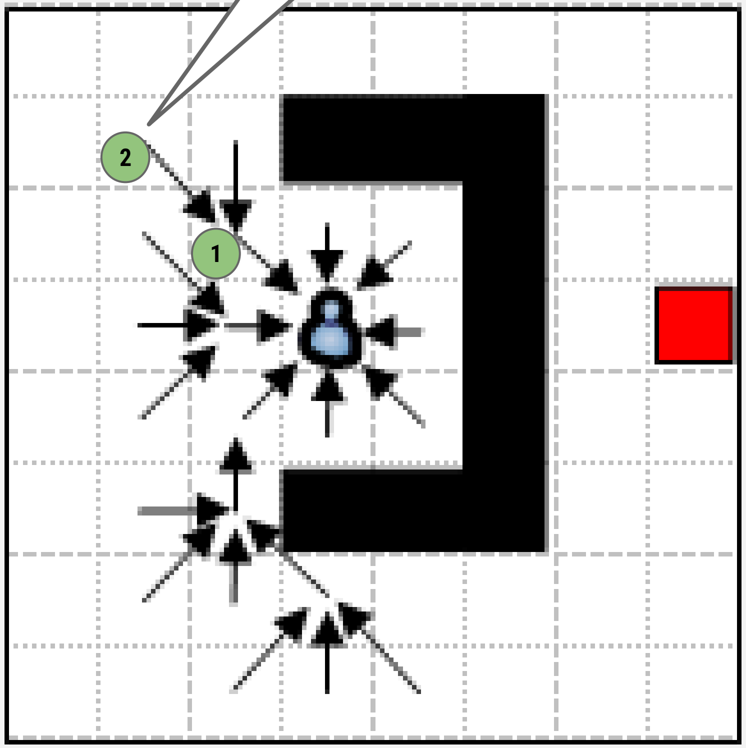
→ Traverses a tree structure by **exploring as far as possible down each branch** before backtracking.



# Pathfinding (Techniques)

➔ Depth-First Search (DFS)

For step 2, select an adjacent cell from the cell selected in step 1.

| STRATEGY | EXAMPLE |
|---|---|
| ➔ Traverses a tree structure by **exploring as far as possible down each branch** before backtracking. |  |

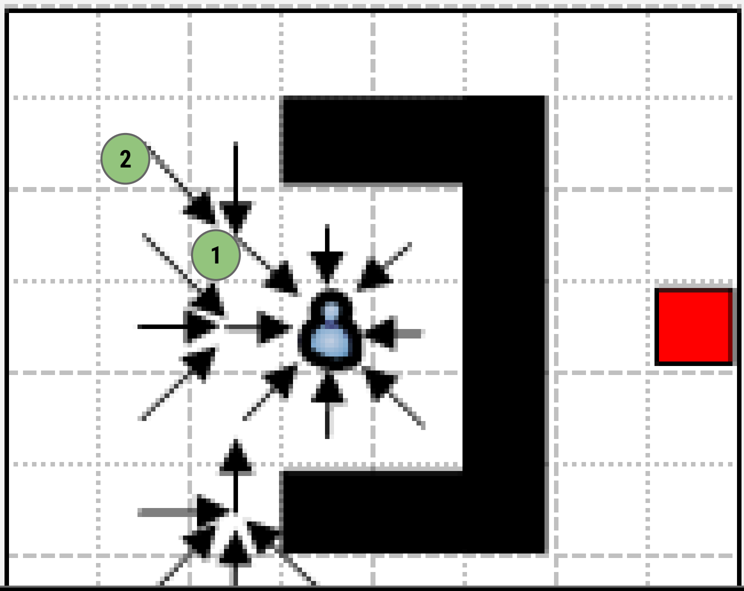
# Pathfinding (Techniques)

➔ Depth-First Search (DFS)

**STRATEGY**

➔ Traverses a tree structure by **exploring as far as possible down each branch** before backtracking.

**EXAMPLE**



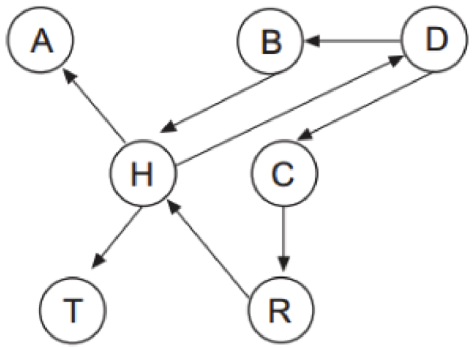
➔ ***Can have problems if the depth or the search is not limited.***

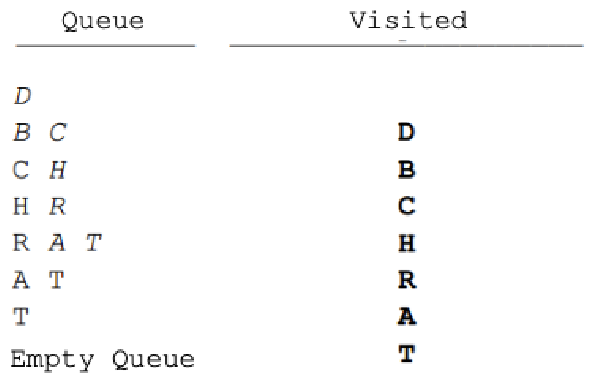
➔ *When using a recursive implementation, may lead to a stack overflow.*

➔ *It's better to implement it iteratively using a stack.*

# Pathfinding (Techniques)

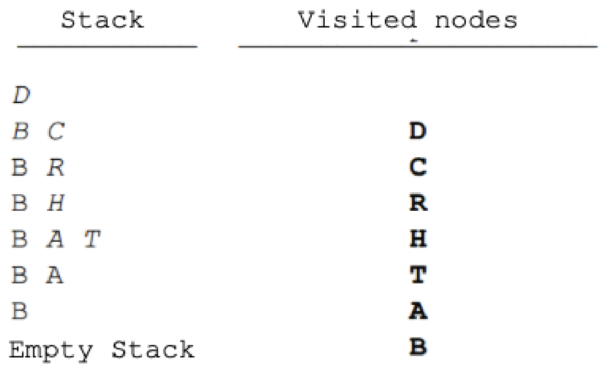
➔ Difference between BFS and DFS





| Queue | Visited |
|---|---|
| D | |
| B C | **D** |
| C H | **B** |
| H R | **C** |
| R A T | **H** |
| A T | **R** |
| T | **A** |
| Empty Queue | **T** |

**BFS**



| Stack | Visited nodes |
|---|---|
| D | |
| B C | **D** |
| B R | **C** |
| B H | **R** |
| B A T | **H** |
| B A | **T** |
| B | **A** |
| Empty Stack | **B** |

**DFS**

## Pathfinding (Techniques)

→ Dijkstra

| STRATEGY | EXAMPLE |
|---|---|
| → **Keeps track of the total cost** from the start to every node that is visited, and uses it to determine the best order to traverse the grapha. | |

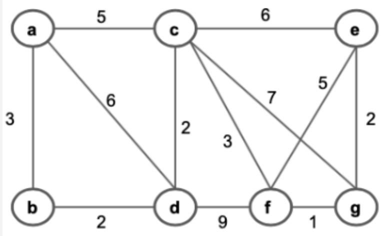
## Pathfinding (Techniques)

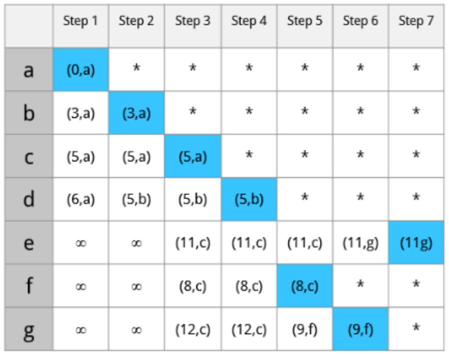
→ Dijkstra

**STRATEGY**

→ **Keeps track of the total cost** from the start to every node that is visited, and uses it to determine the best order to traverse the graph.

**EXAMPLE**





| | Step 1 | Step 2 | Step 3 | Step 4 | Step 5 | Step 6 | Step 7 |
|---|---|---|---|---|---|---|---|
| a | (0,a) | * | * | * | * | * | * |
| b | (3,a) | (3,a) | * | * | * | * | * |
| c | (5,a) | (5,a) | (5,a) | * | * | * | * |
| d | (6,a) | (5,b) | (5,b) | (5,b) | * | * | * |
| e | ∞ | ∞ | (11,c) | (11,c) | (11,c) | (11,g) | (11g) |
| f | ∞ | ∞ | (8,c) | (8,c) | (8,c) | * | * |
| g | ∞ | ∞ | (12,c) | (12,c) | (9,f) | (9,f) | * |

→ *Works with weighted graphs and return the shortest path.*

→ ***Might involve a lot of searching.***

## Pathfinding (Techniques)

➔ Best First Search

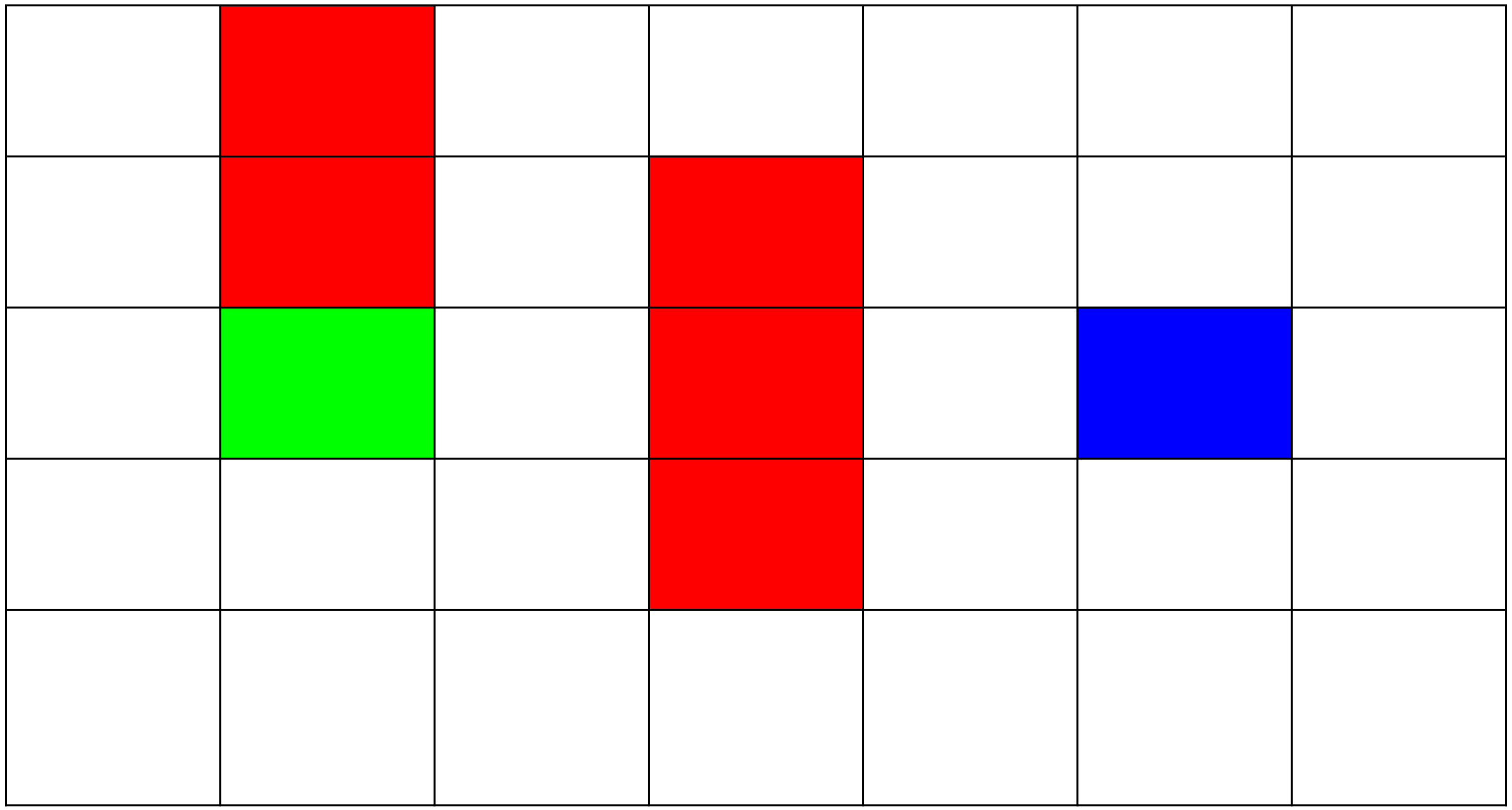
| STRATEGY | EXAMPLE |
|---|---|
| ➔ Similar to BFS but uses an heuristic that chooses the most promising neighbor first.<br><br>➔ The path returned may not be the shortest, but it’s faster to run than BFS. | A* |

## Pathfinding (A*)

| STRATEGY | CHARACTERISTICS |
|---|---|
| ➔ Similar to Dijkstra but uses an **heuristic that to estimate how likely each node is close to the goal**, in order to make the best decision. | ➔ **Precise**.<br>➔ **Efficient and Fast**. Finds the shortest path in a weighted graph processing fewer cells and taking less time to perform the calculation.<br>➔ Maybe doesn't find the optimal path, but **will find a close option**.<br>➔ **One of the most commonly used in games programming**. It's extremely configurable to the particular type of game and map. |

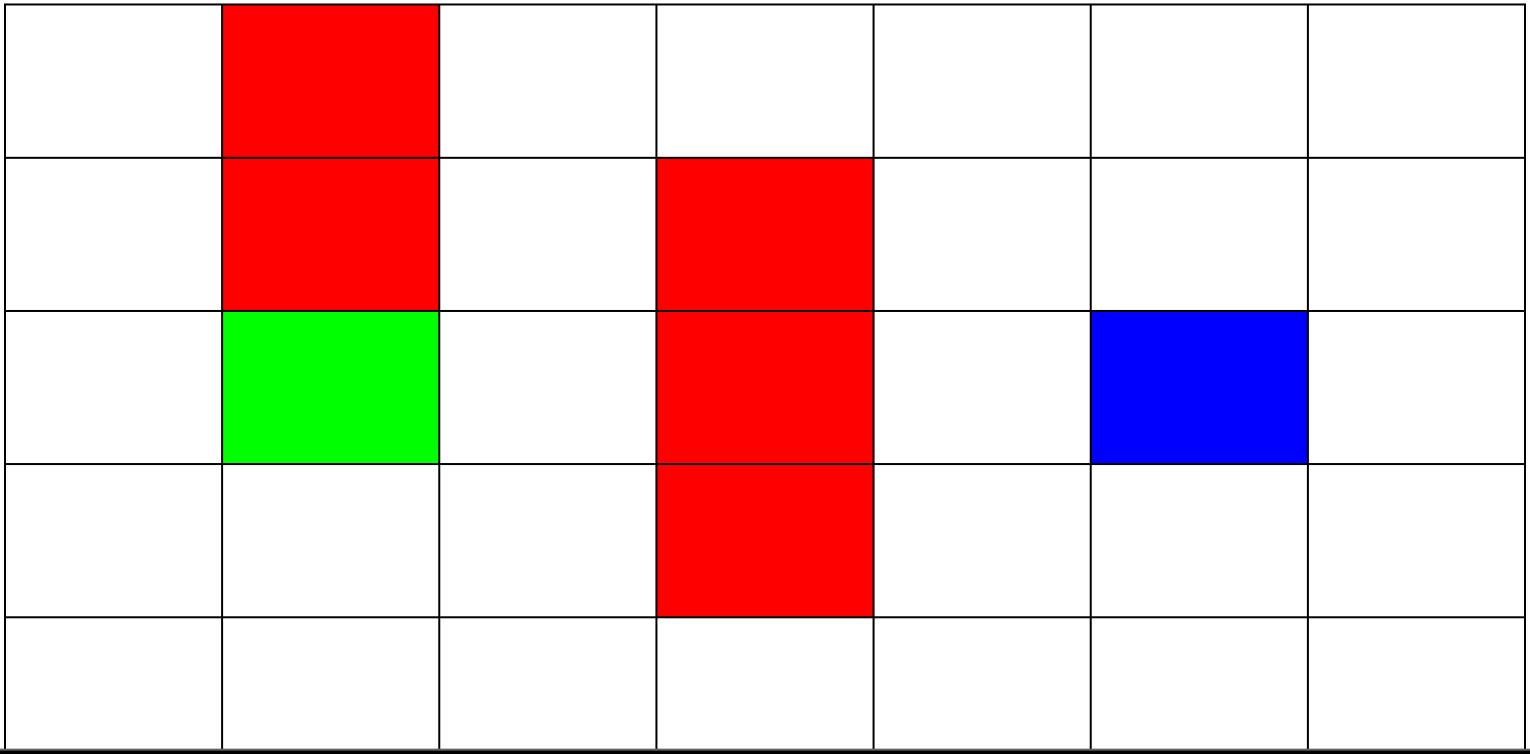
## Pathfinding (A*)

➔ Search area



## Pathfinding (A*)

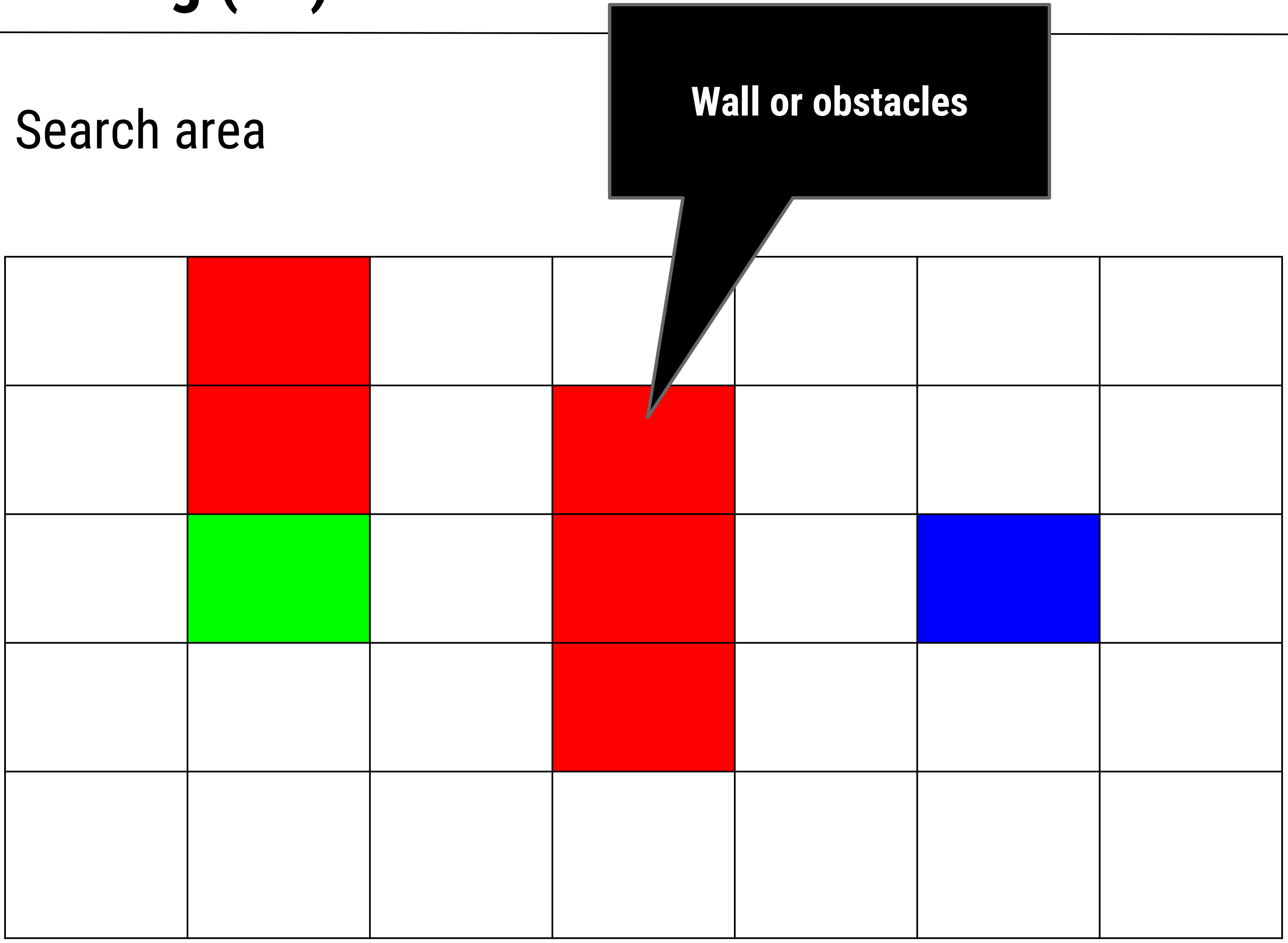
➔ Search area



➔ *Divide the area into a square grid. You can divide the area in another kind of shape.*
➔ *Center point in each square is called node.*
➔ *Each cell has a state*

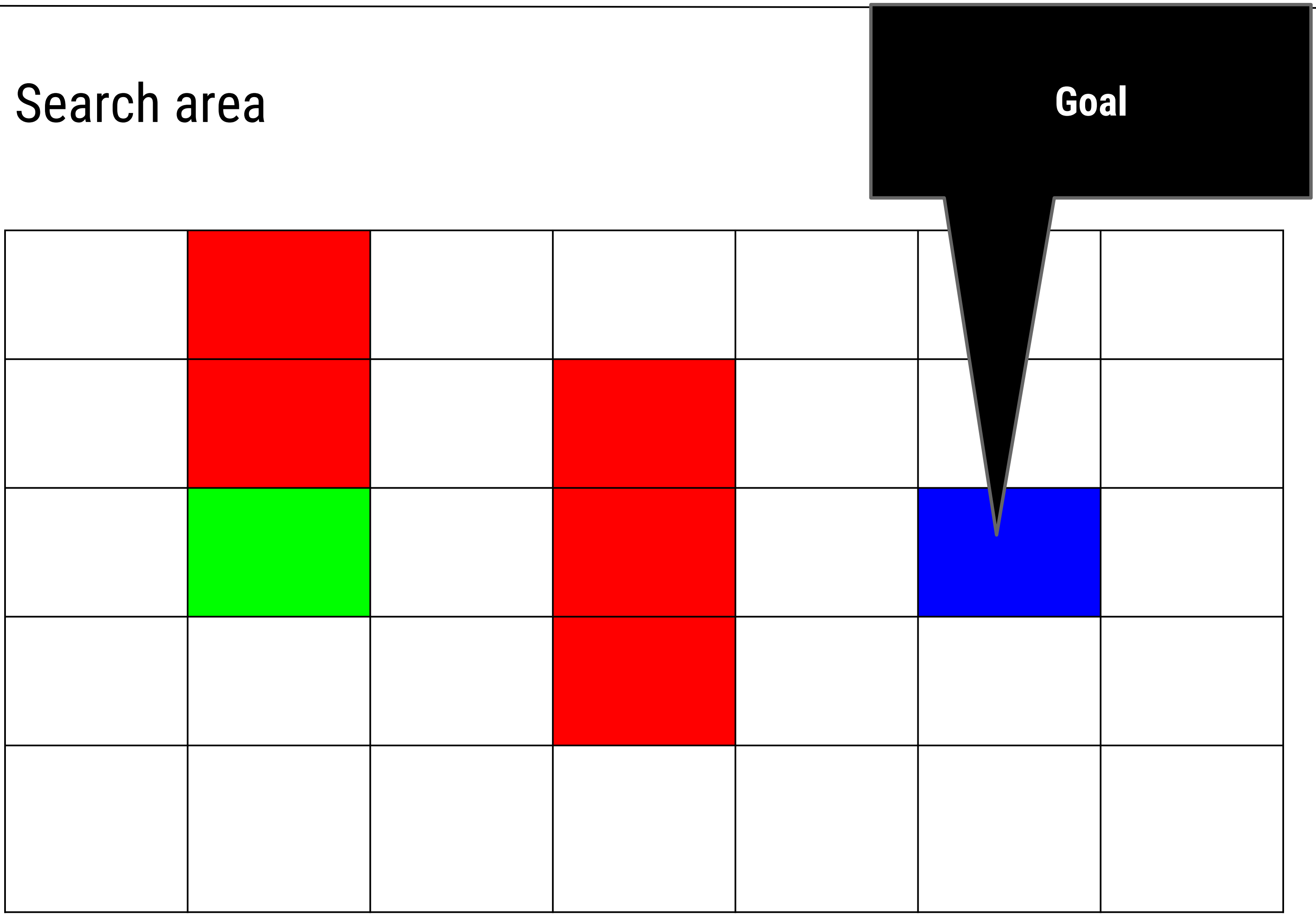
## Pathfinding (A*)

➔ Search area



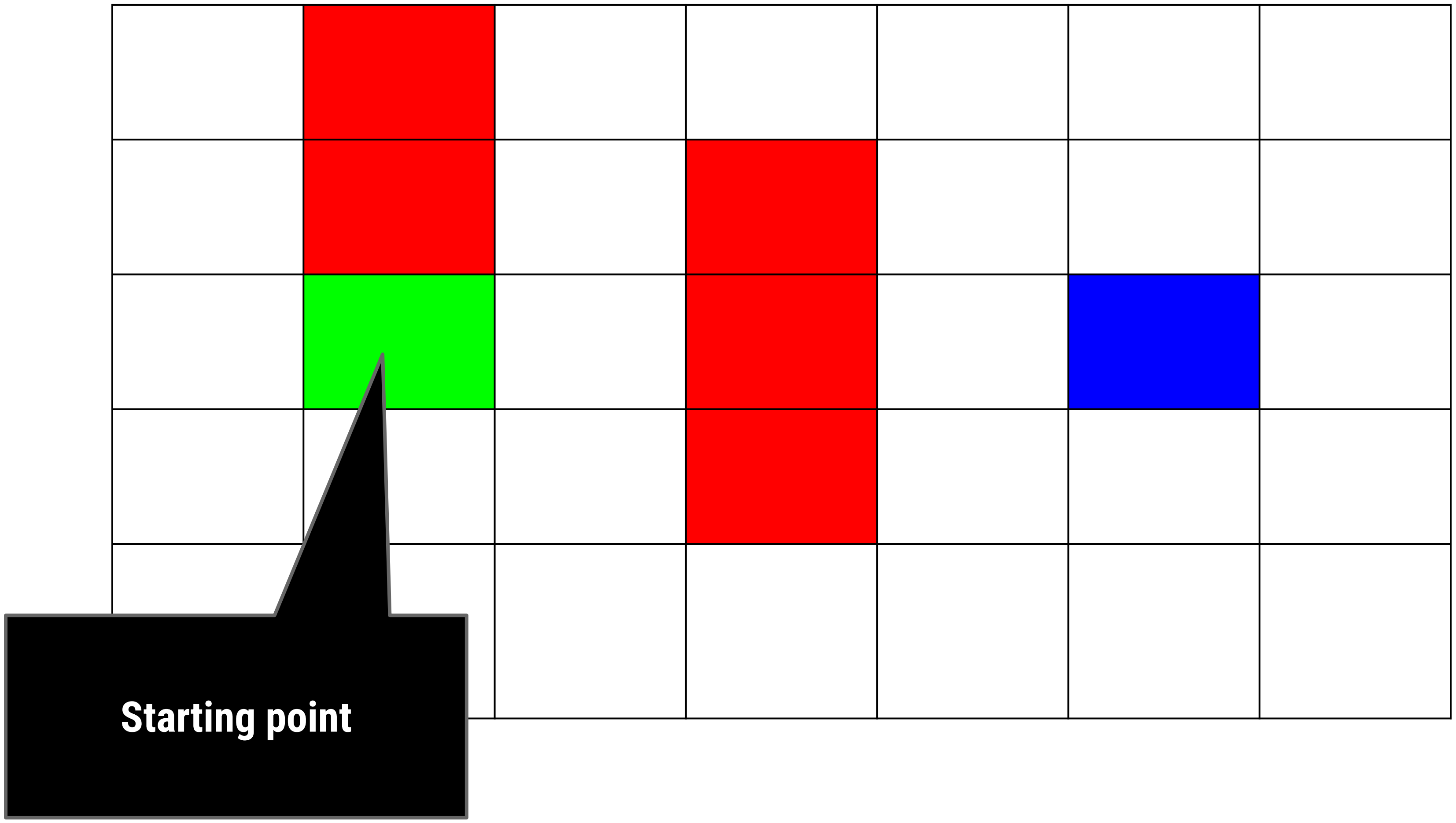
# Pathfinding (A*)

➔ Search area



# Pathfinding (A*)

➔ Search area



## Pathfinding (A*)

➔ Algorithm

1. Add the starting point to an open-list of elements.

1. Check the adjacent walkable squares, add the squares to the open-list, save the starting point as a parent of these squares.

1. Remove the starting point from the open-list and add it to the close-list.

1. Get one of the adjacent squares in the open-list and repeat the process.

## Pathfinding (A*)

➔ Path Scoring

$$F = G + H$$

G = movement cost to move from the starting point to a given square on the grid.

H = estimated cost to move from a given square on the grid to the final destination (heuristic).

## Pathfinding (A*)

➔ Path Scoring

$$F = G + H$$

G = movement cost to move from the st
square on the grid.

H = estimated cost to move from a give
to the final destination (heuristic).

➔ You have to choose the cell with the lowest F score.

➔ You have to assign a cost to move horizontal, vertical and diagonal.

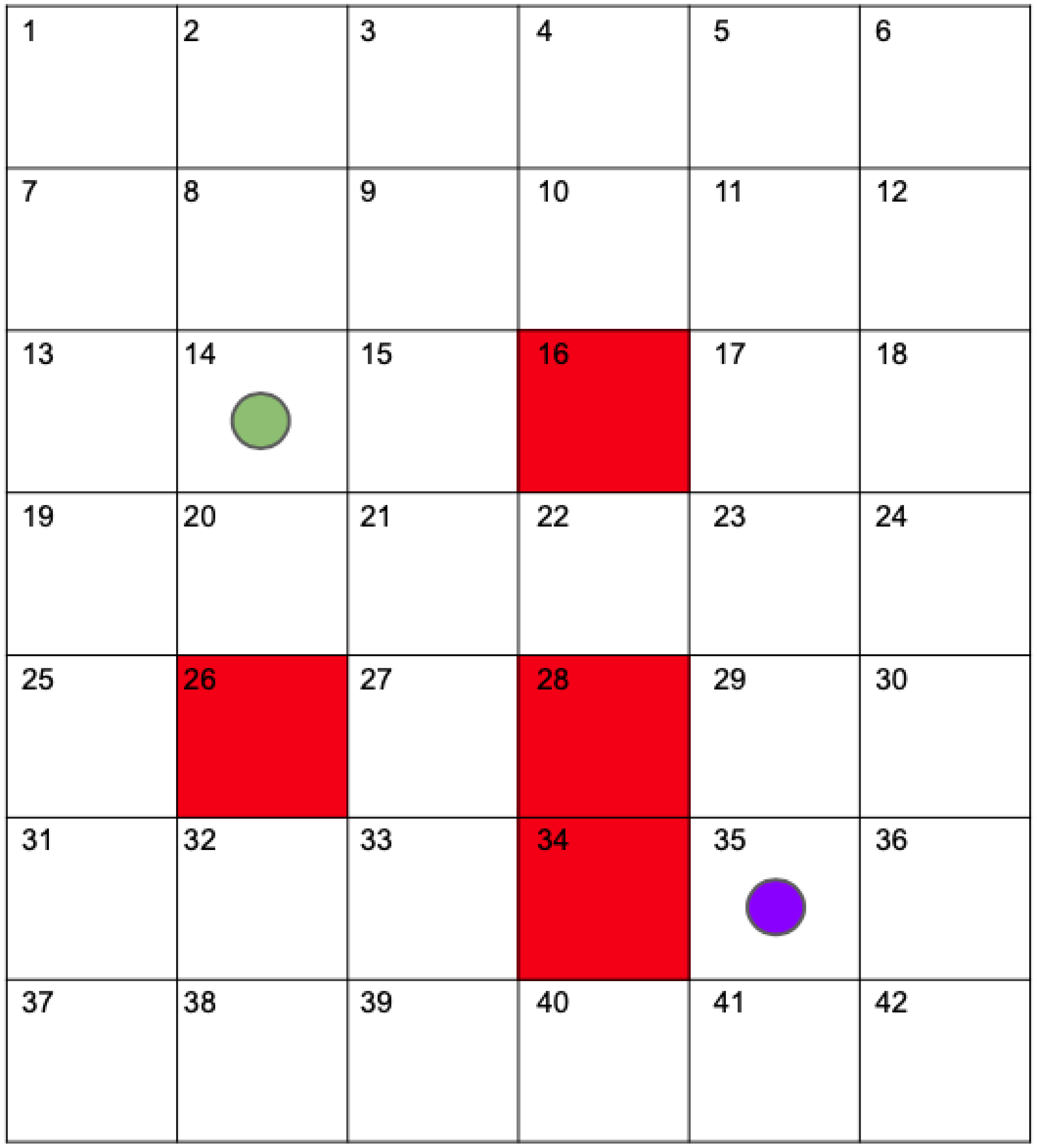
## Pathfinding (A*)

➔ How to calculate H

Variety of methods, there isn’t a well known solution. For Example:

| EUCLIDEAN DISTANCE | MANHATTAN DISTANCE |
|---|---|
| dx = targetX - currentX;<br>dy = targetY - currentY;<br>h = sqrt((dx * dx) + (dy * dy)); | dx = abs(targetX - currentX);<br>dy = abs(targetY - currentY);<br>h = dx + dy; |

# Pathfinding (A*)



## Node Data

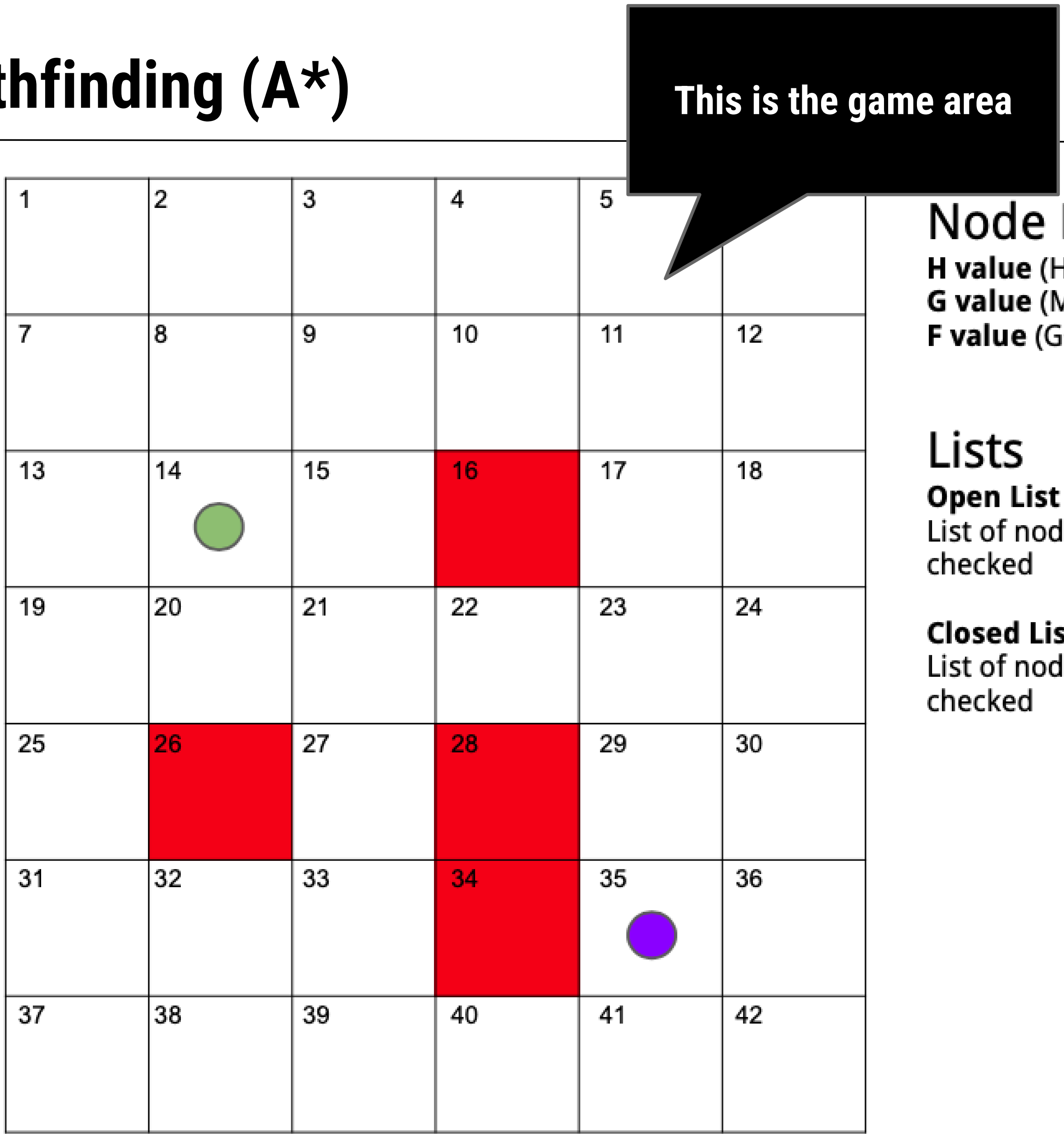
**H value** (Heuristic)
**G value** (Movement cost)
**F value** (G+H)

## Lists

**Open List**
List of nodes that need to be checked

**Closed List**
List of nodes that have been checked

# Pathfinding (A*)



## Node Data

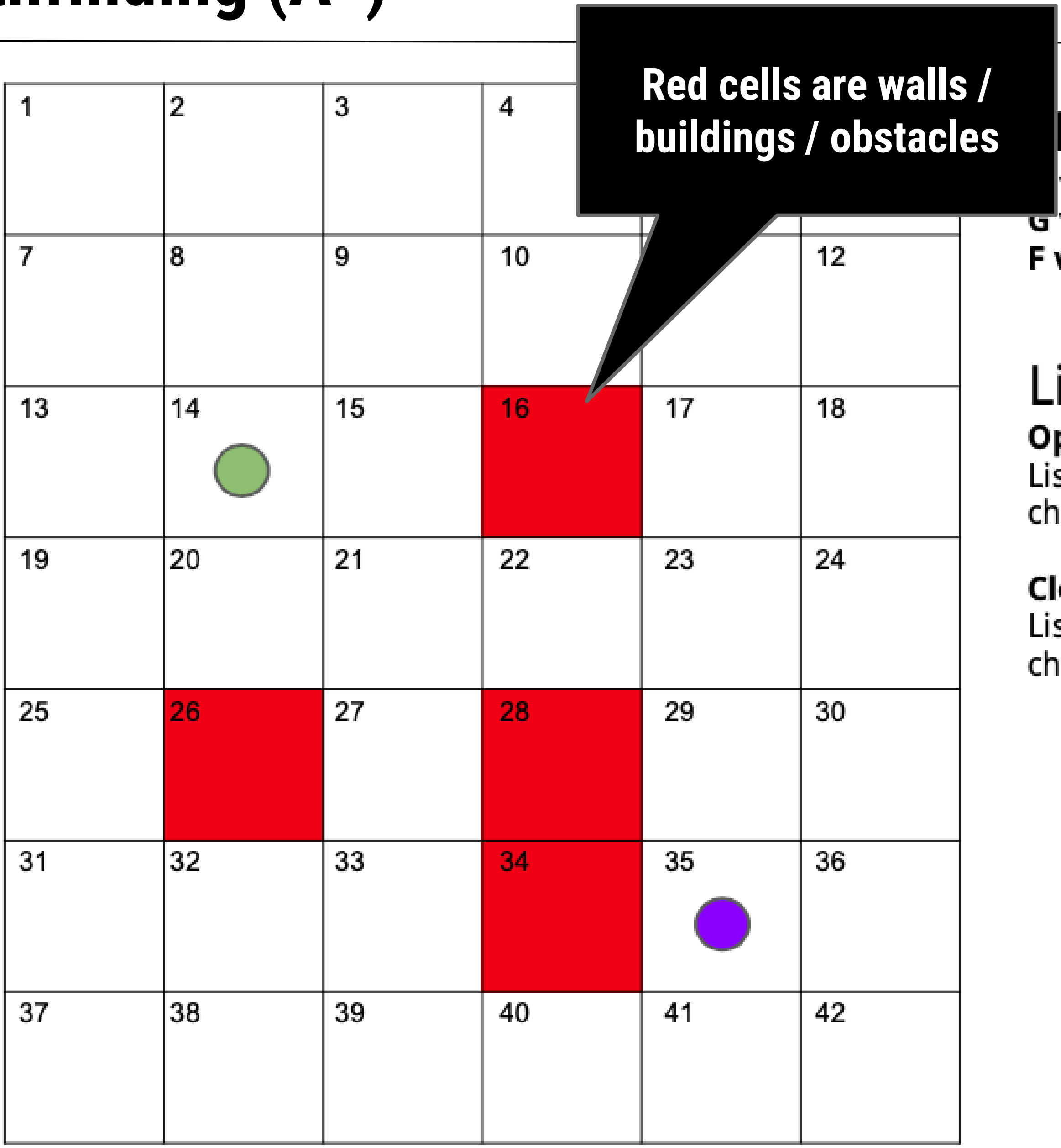
**H value** (Heuristic)
**G value** (Movement cost)
**F value** (G+H)

## Lists

**Open List**
List of nodes that need to be checked

**Closed List**
List of nodes that have been checked

# Pathfinding (A*)



## lode Data

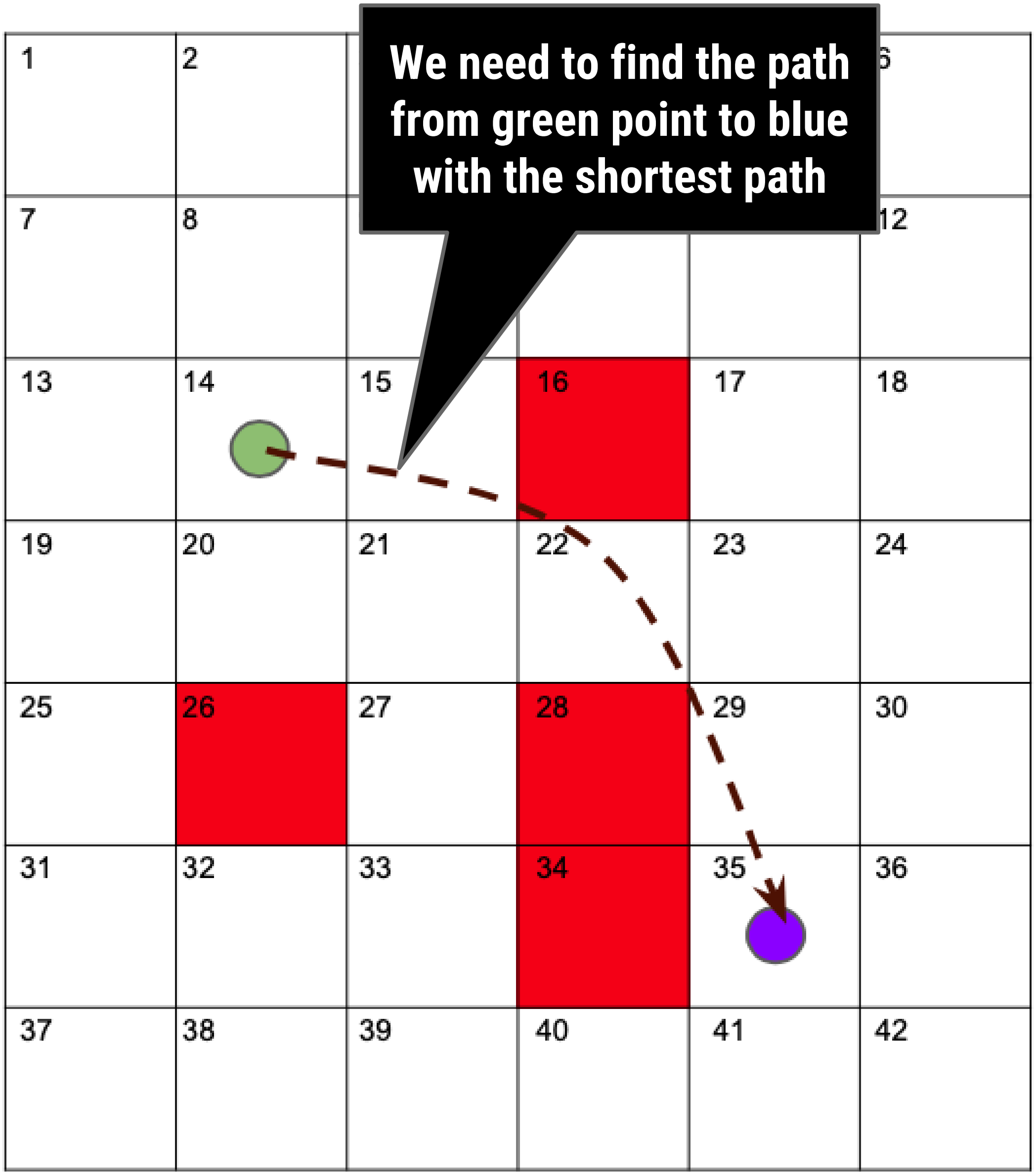
**value** (Heuristic)
**G value** (Movement cost)
**F value** (G+H)

## Lists

**Open List**
List of nodes that need to be checked

**Closed List**
List of nodes that have been checked

# Pathfinding (A*)


We need to find the path from green point to blue with the shortest path
1
2
6
7
8
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42

## Node Data

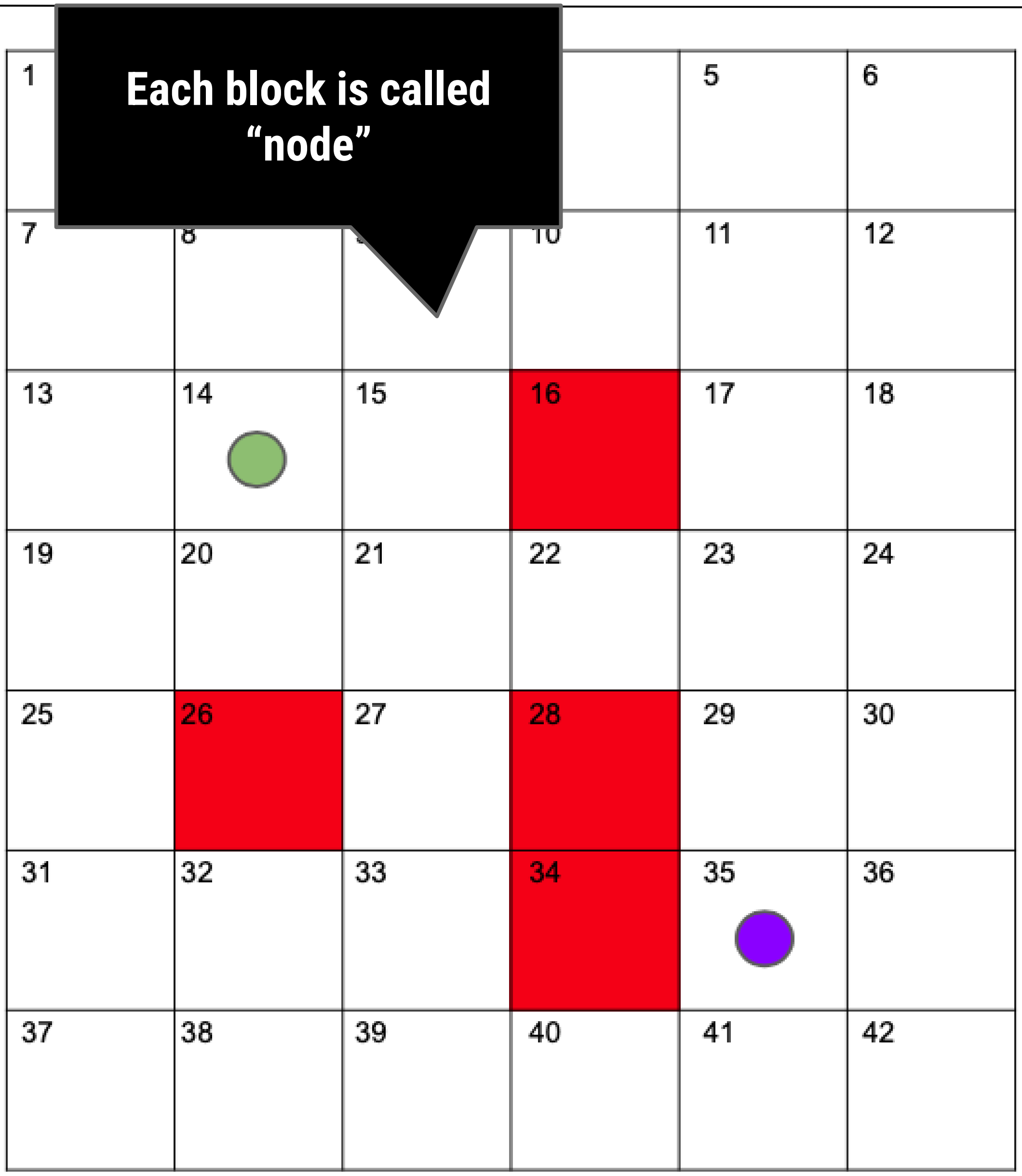
**H value** (Heuristic)
**G value** (Movement cost)
**F value** (G+H)

## Lists

**Open List**
List of nodes that need to be checked

**Closed List**
List of nodes that have been checked

# Pathfinding (A*)



## Node Data

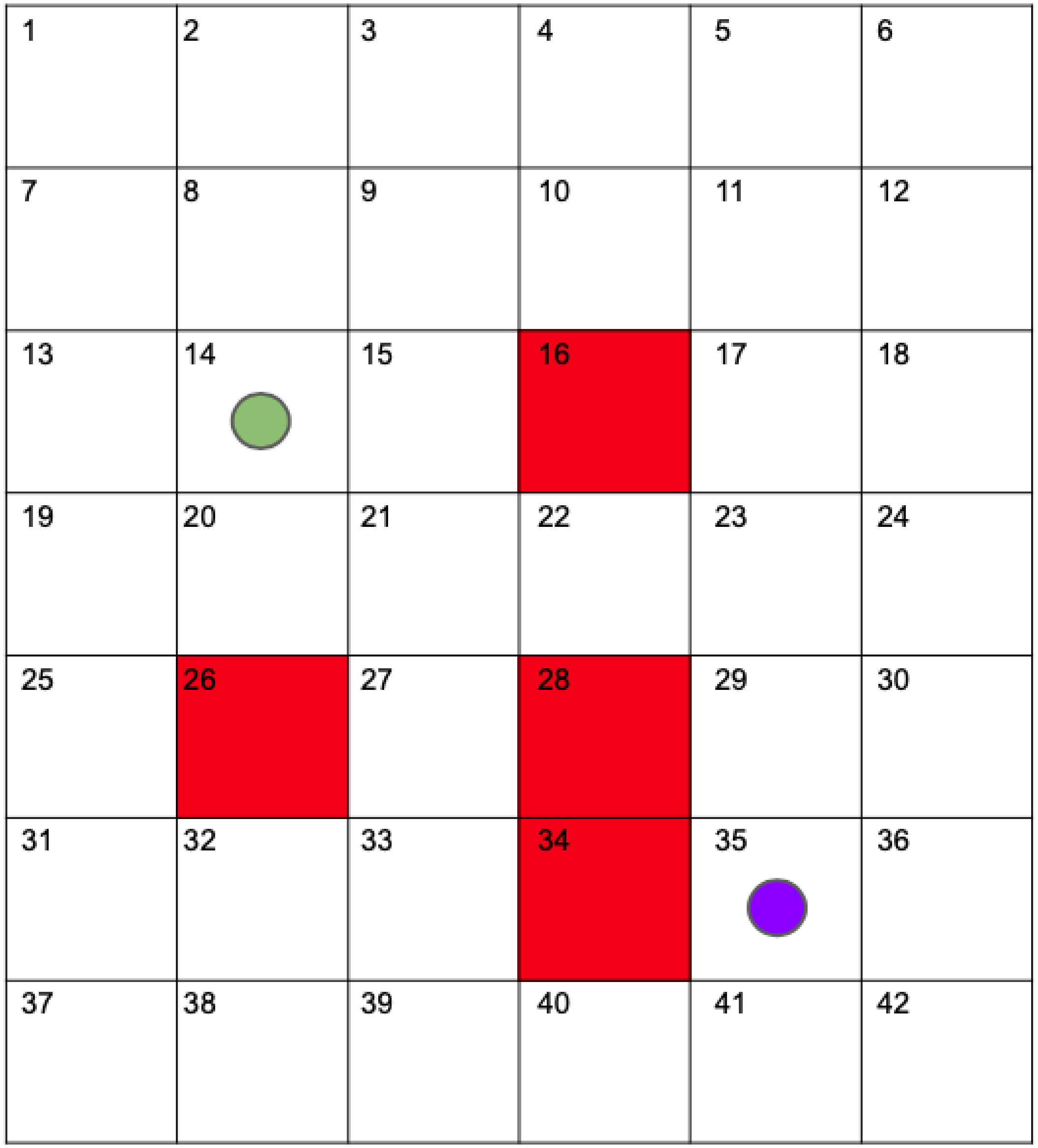
**H value** (Heuristic)
**G value** (Movement cost)
**F value** (G+H)

## Lists

**Open List**
List of nodes that need to be checked

**Closed List**
List of nodes that have been checked

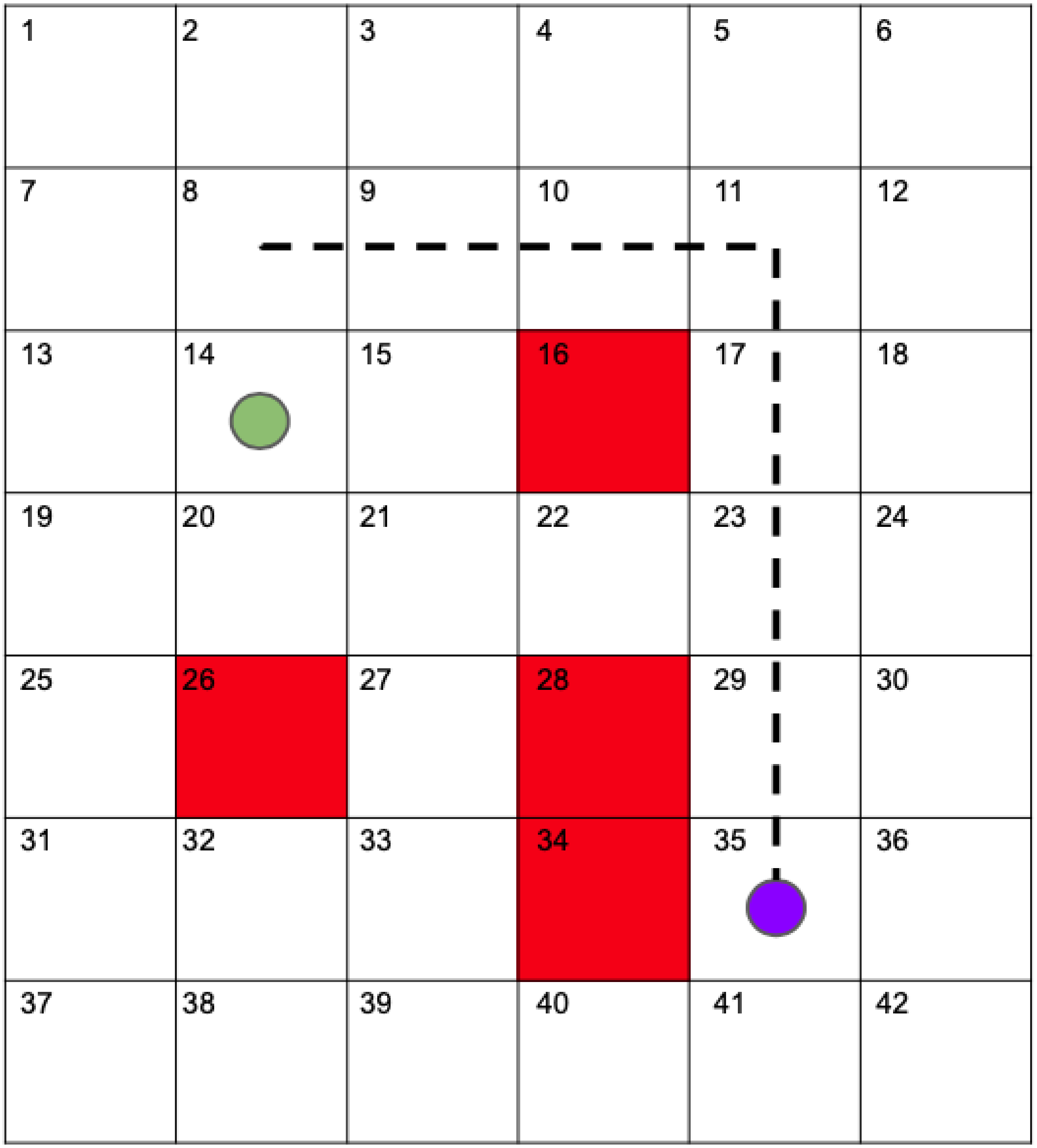
## Pathfinding (A*)



### Heuristic

Using the manhattan formula, calculate the H value for **all nodes**

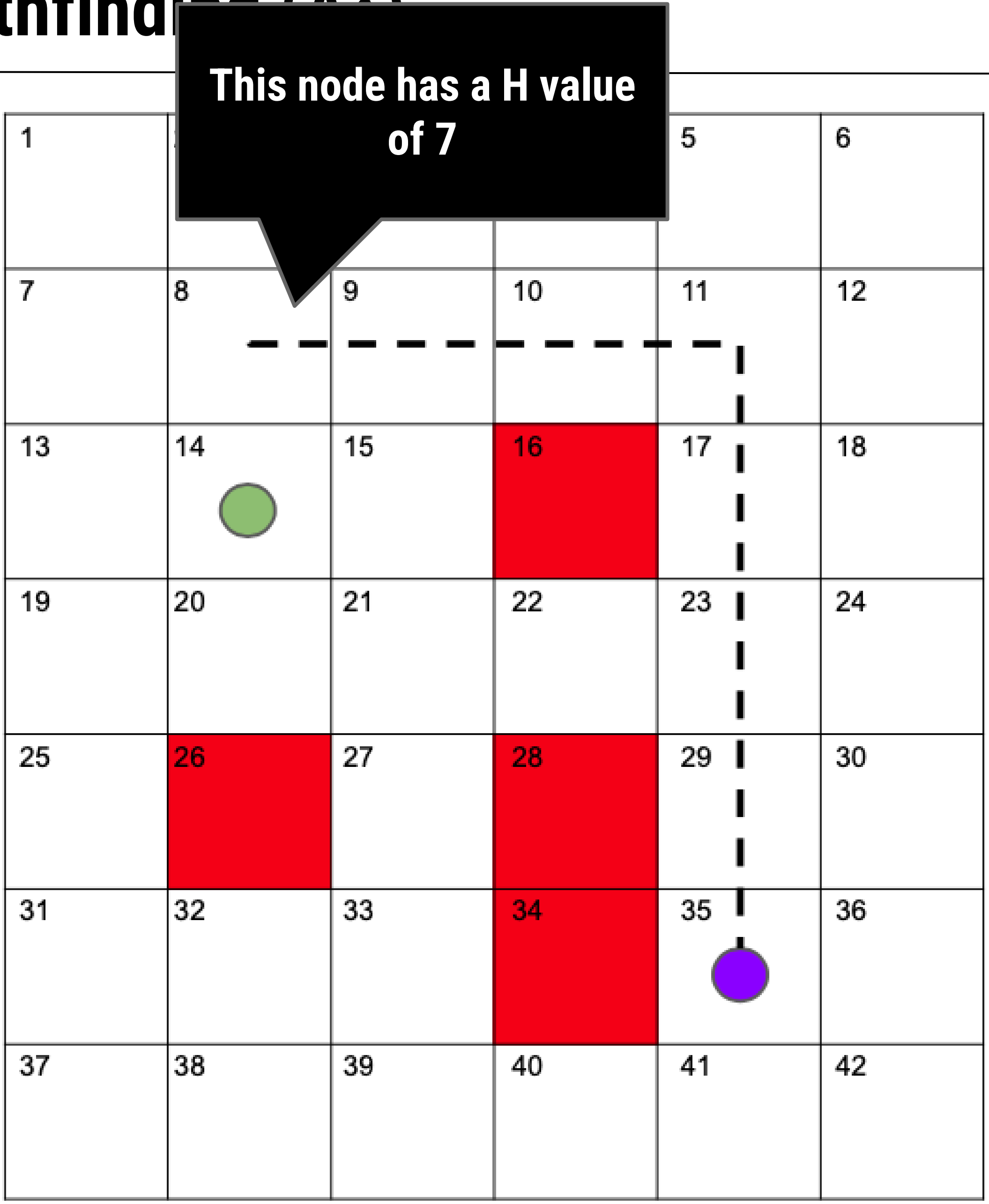
# Pathfinding (A*)



## Heuristic

Using the manhattan formula, calculate the H value for **all nodes**

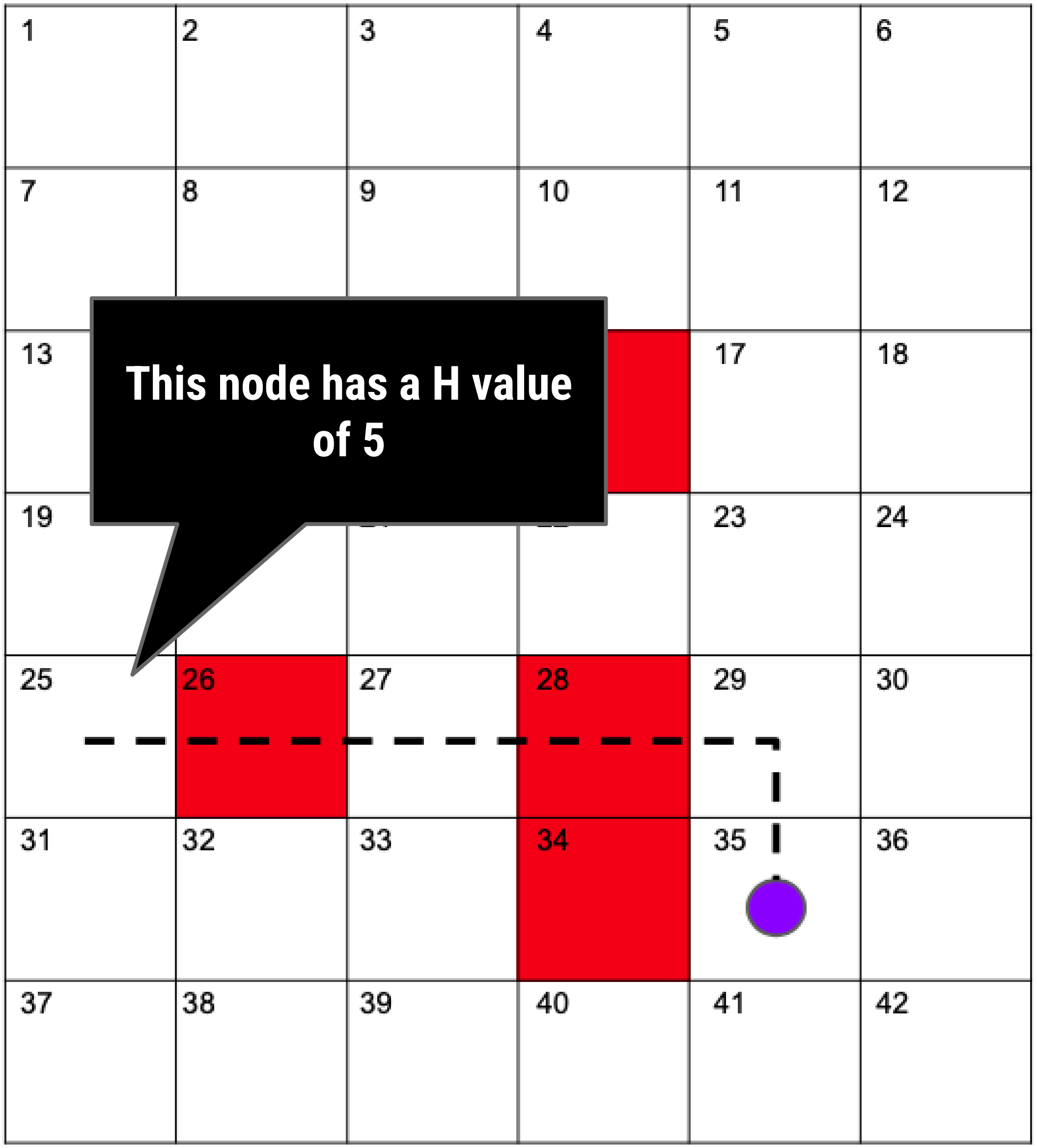
# Pathfinding (A*)



## Heuristic

Using the manhattan formula, calculate the H value for **all nodes**

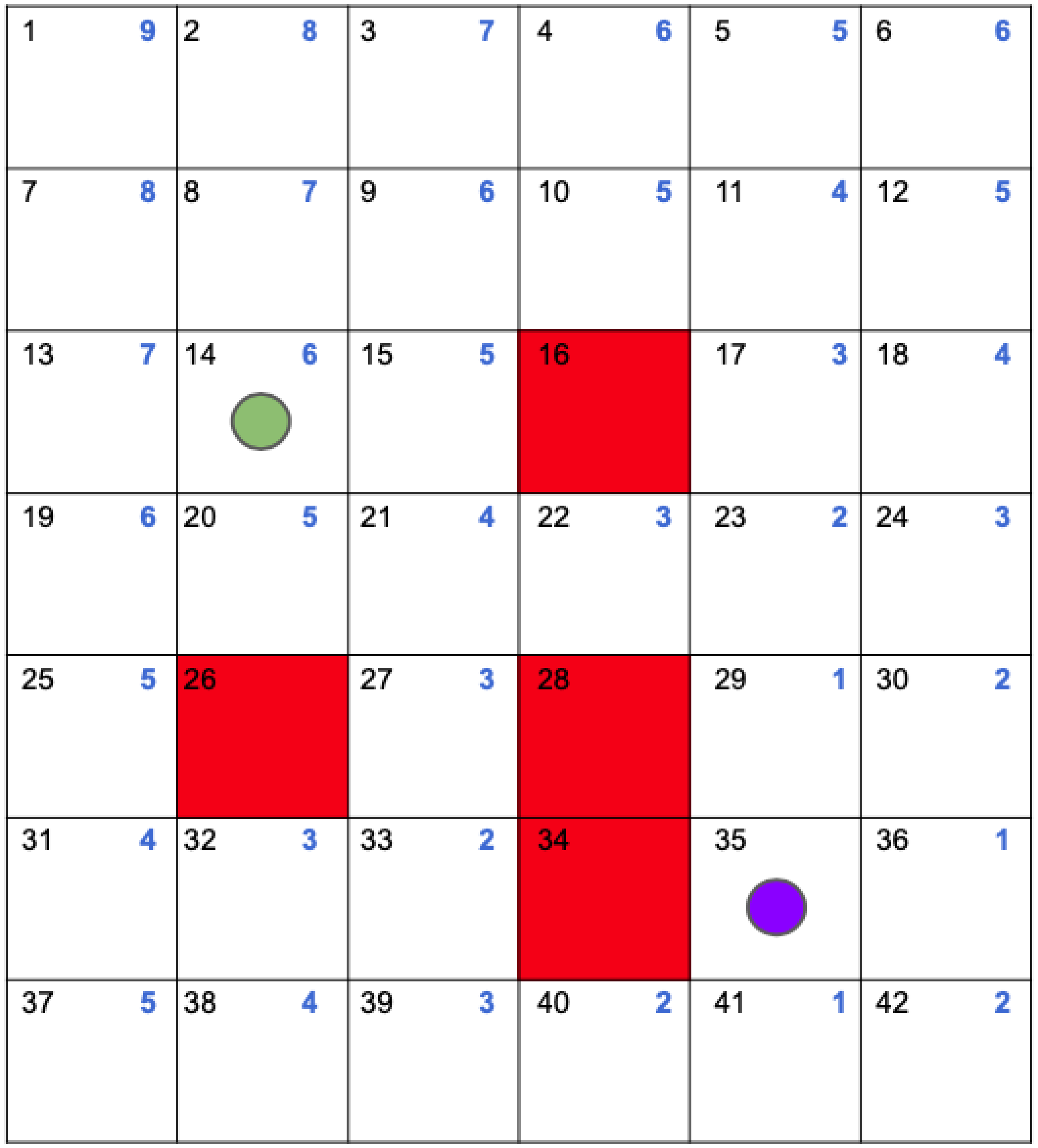
# Pathfinding (A*)



## Heuristic

Using the manhattan formula, calculate the H value for **all nodes**

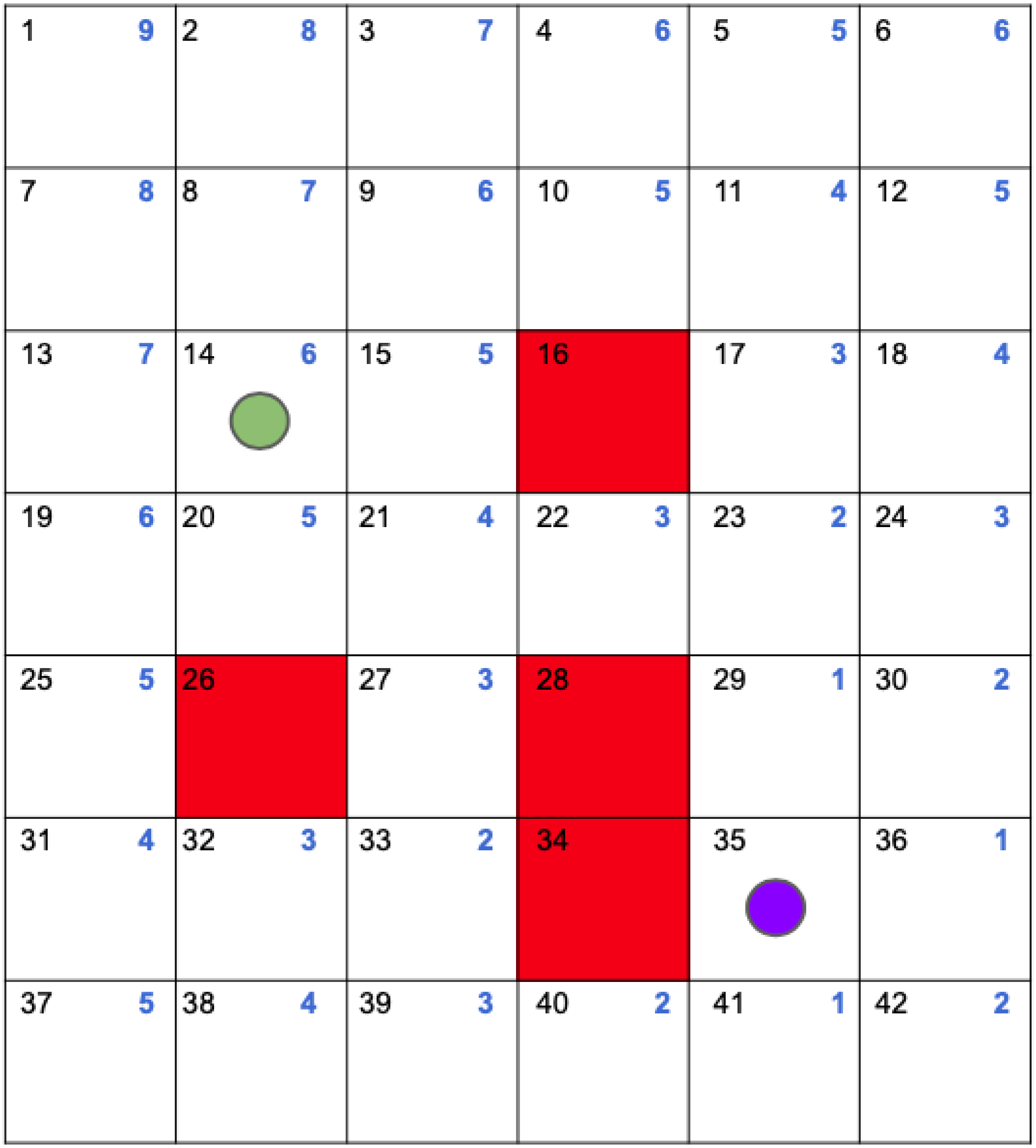
# Pathfinding (A*)



## Heuristic

Using the manhattan formula, calculate the H value for **all nodes**

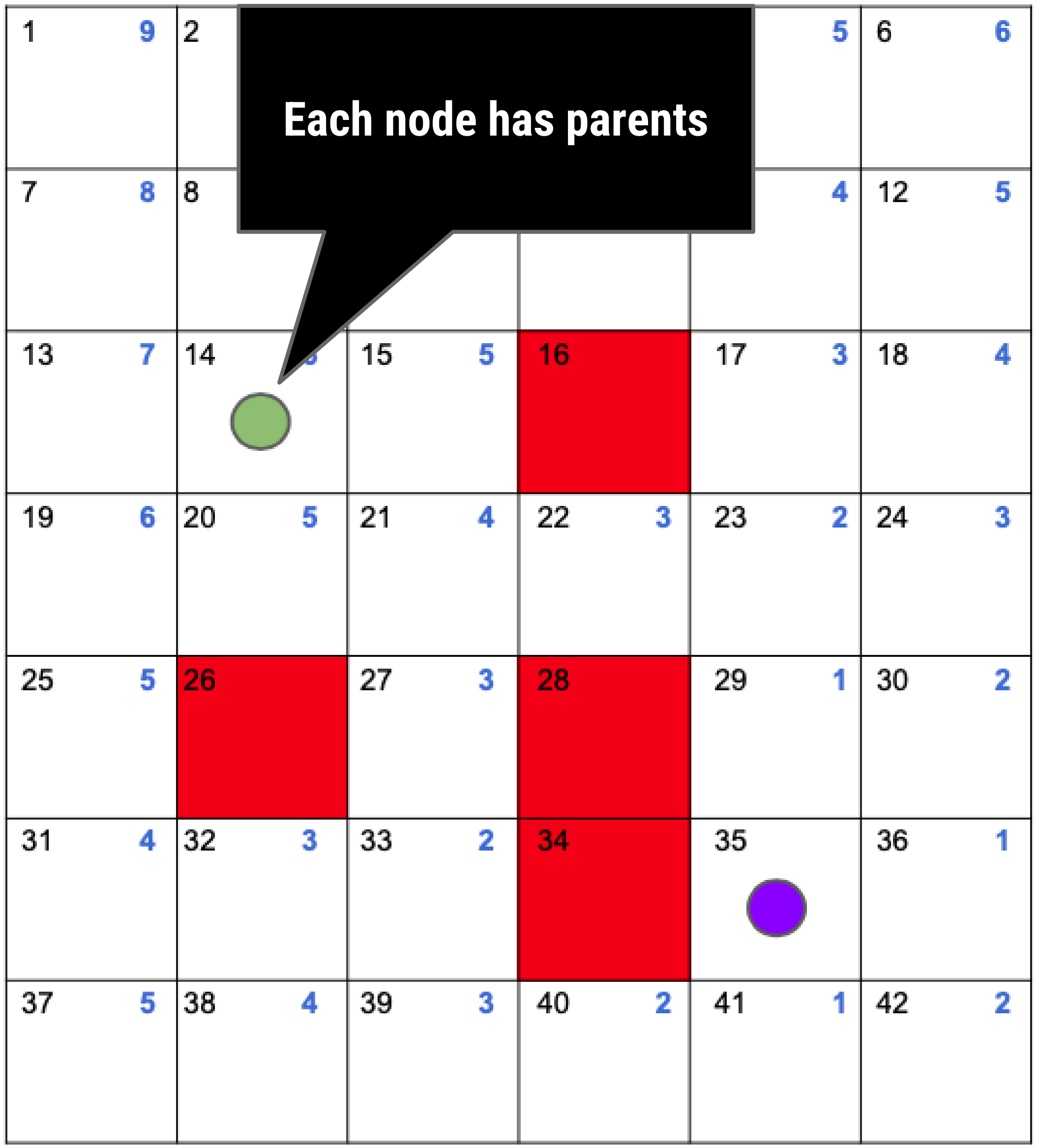
# Pathfinding (A*)



## Open List

## Closed List

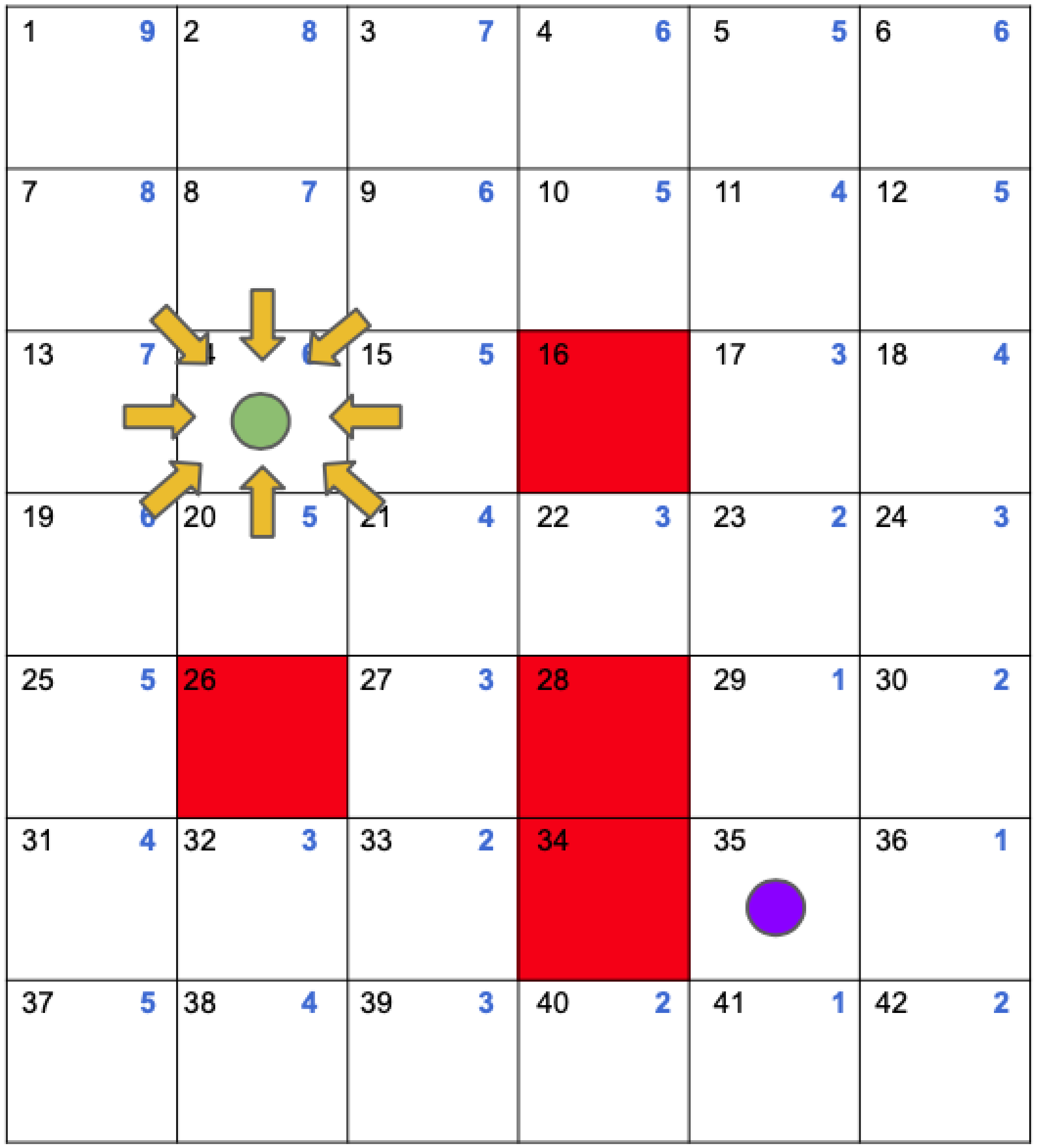
# Pathfinding (A*)



## G Value

You should define a consistent formula for this. For example: **horizontal and vertical is 10, diagonal is 14**

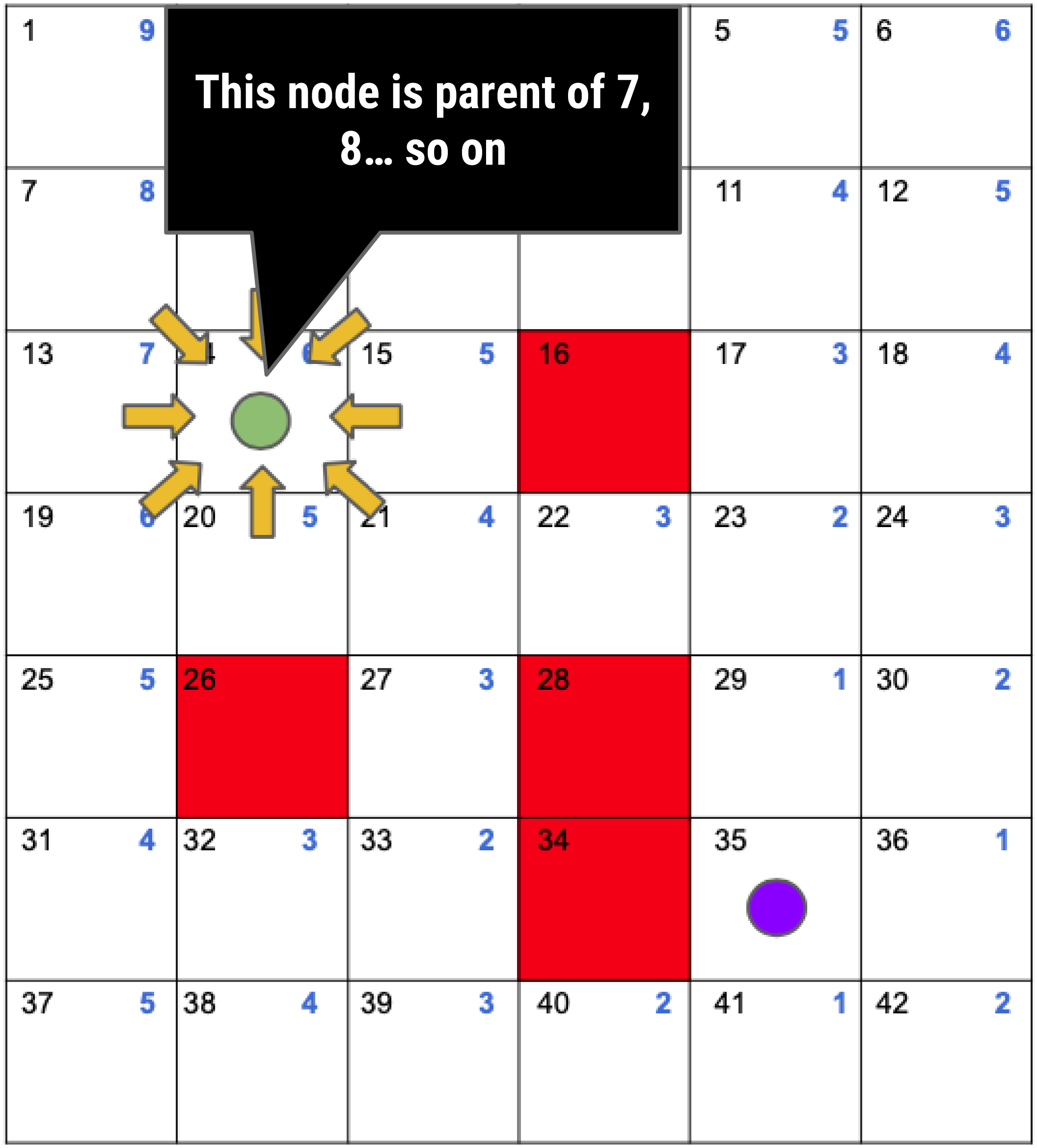
## Pathfinding (A*)



**Open List**
7,8,9,13,15,19,20,21

**Closed List**
14

# Pathfinding (A*)



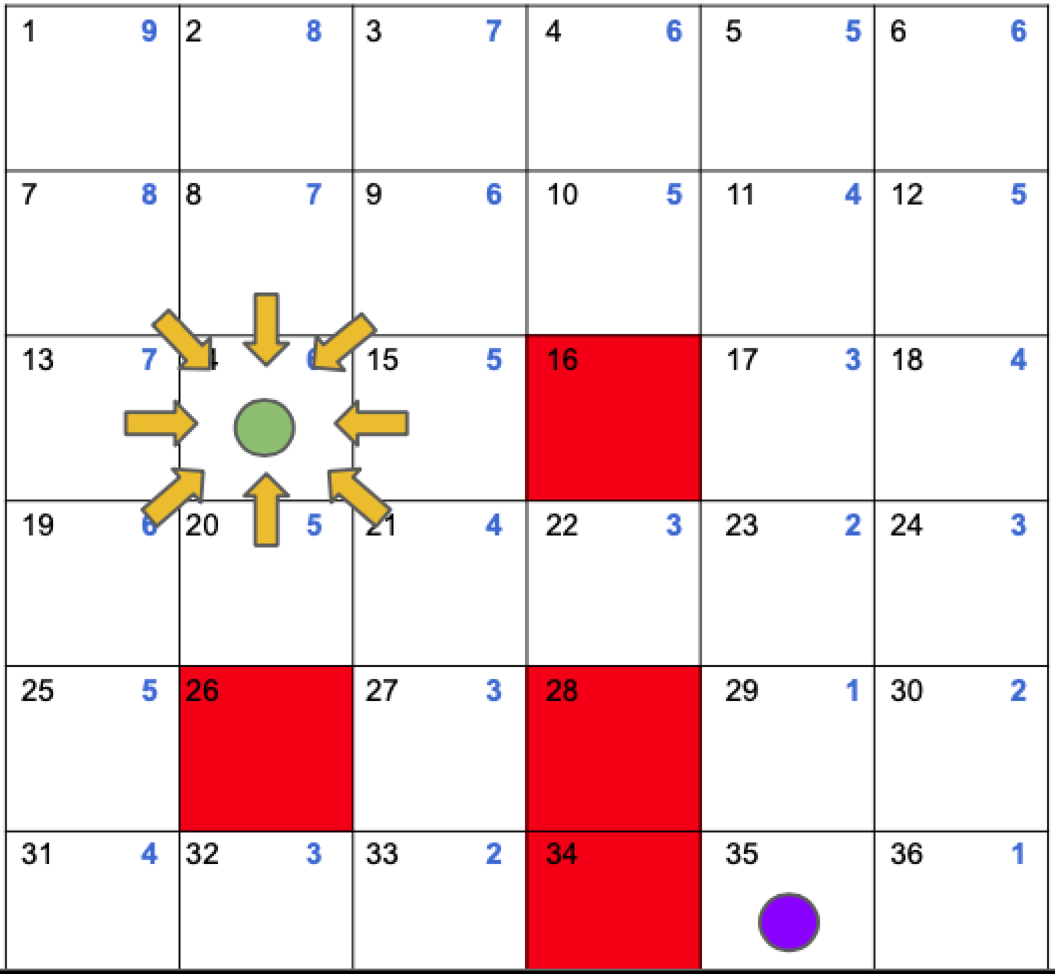
**Open List**

7,8,9,13,15,19,20,21

**Closed List**

14

# Pathfinding (A*)



## Open List

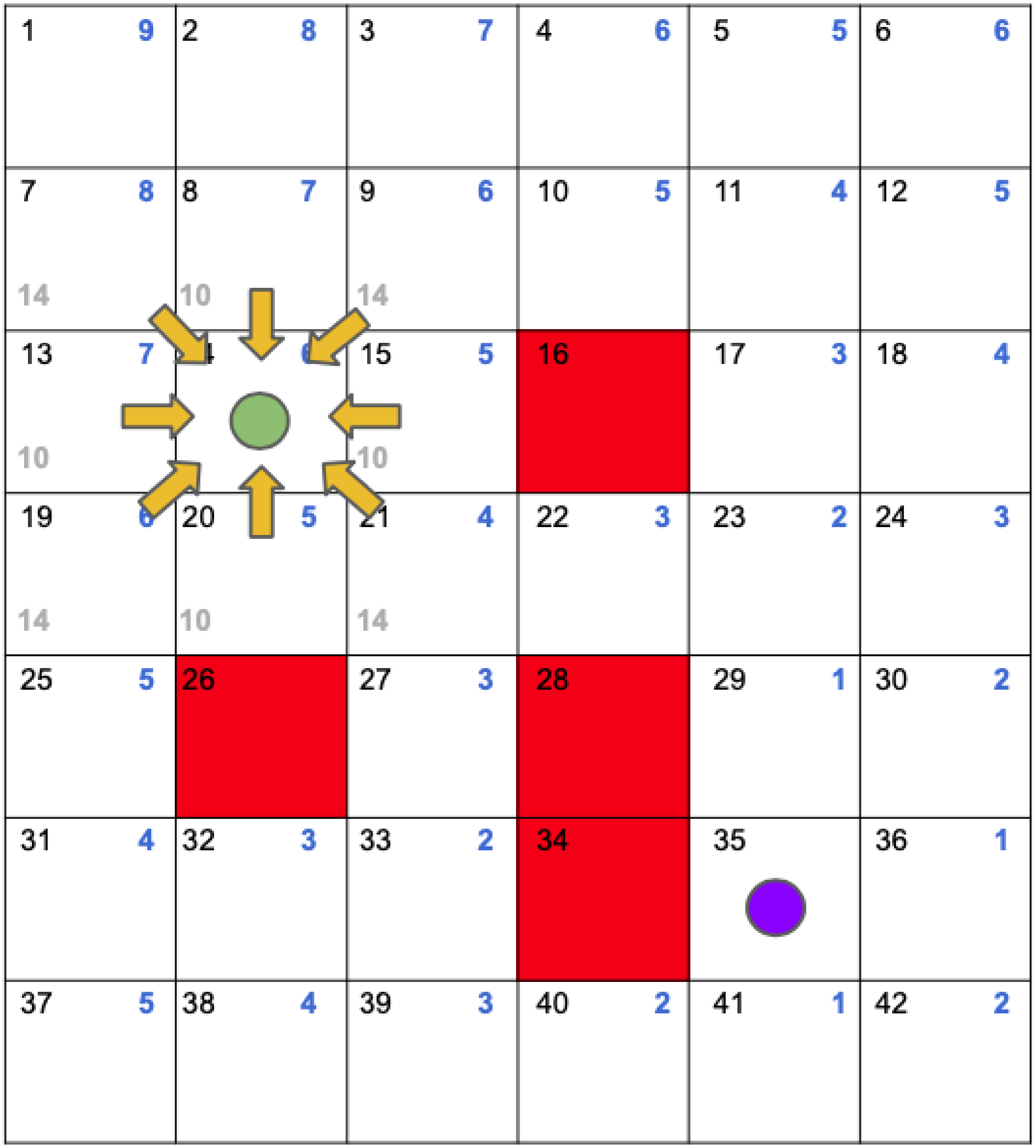
7,8,9,13,15,19,20,21

## Closed List

14

**Now, calculate G**

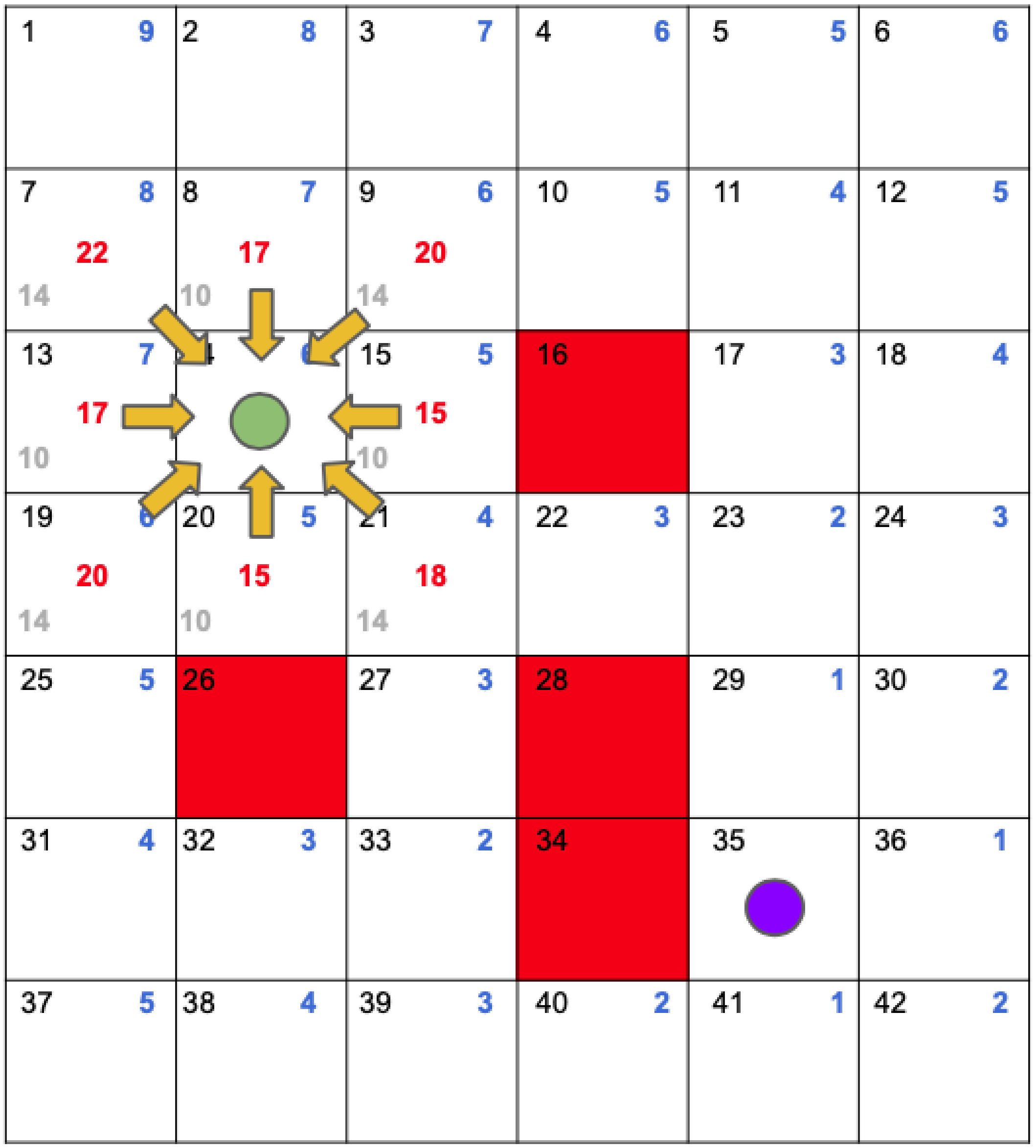
# Pathfinding (A*)



## Open List
7,8,9,13,15,19,20,21

## Closed List
14

## Pathfinding (A*)



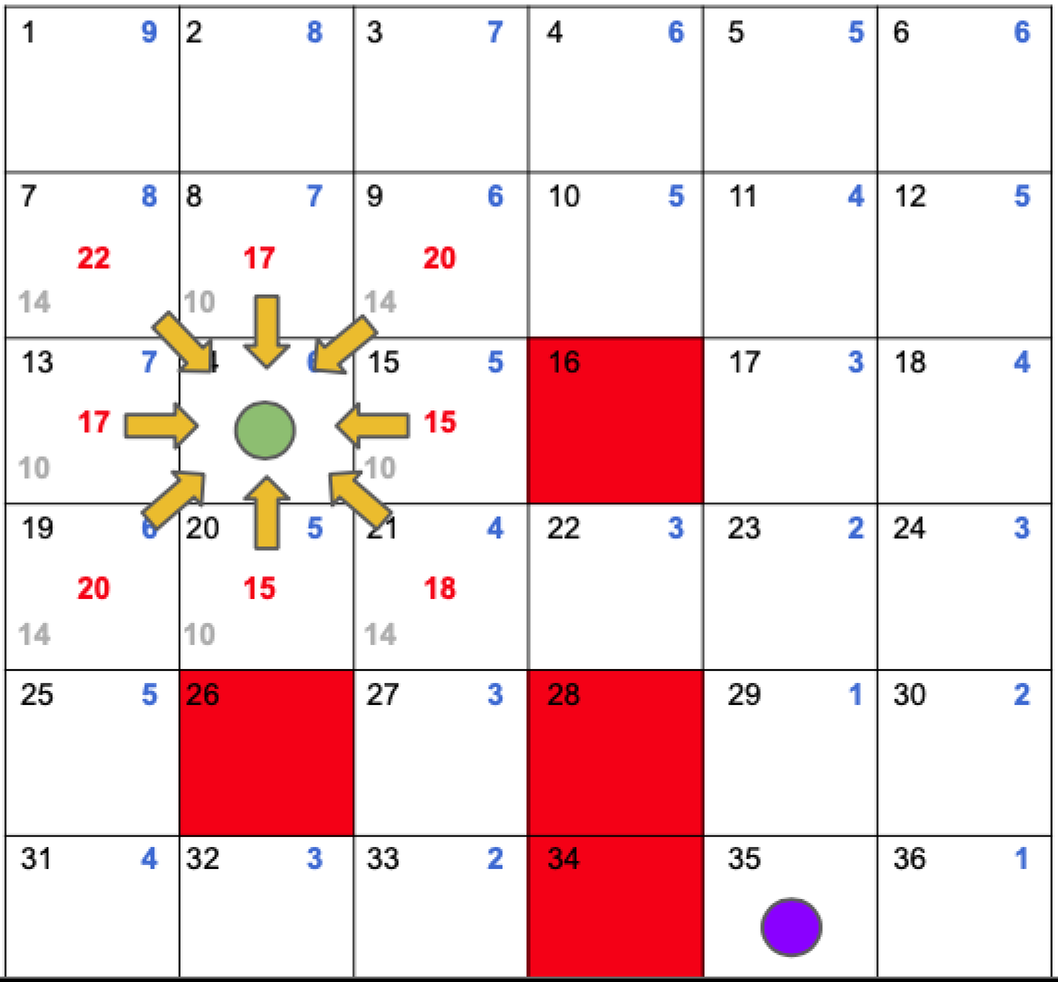
**Open List**

7,8,9,13,15,19,20,21

**Closed List**

14

## Pathfinding (A*)



**Open List**

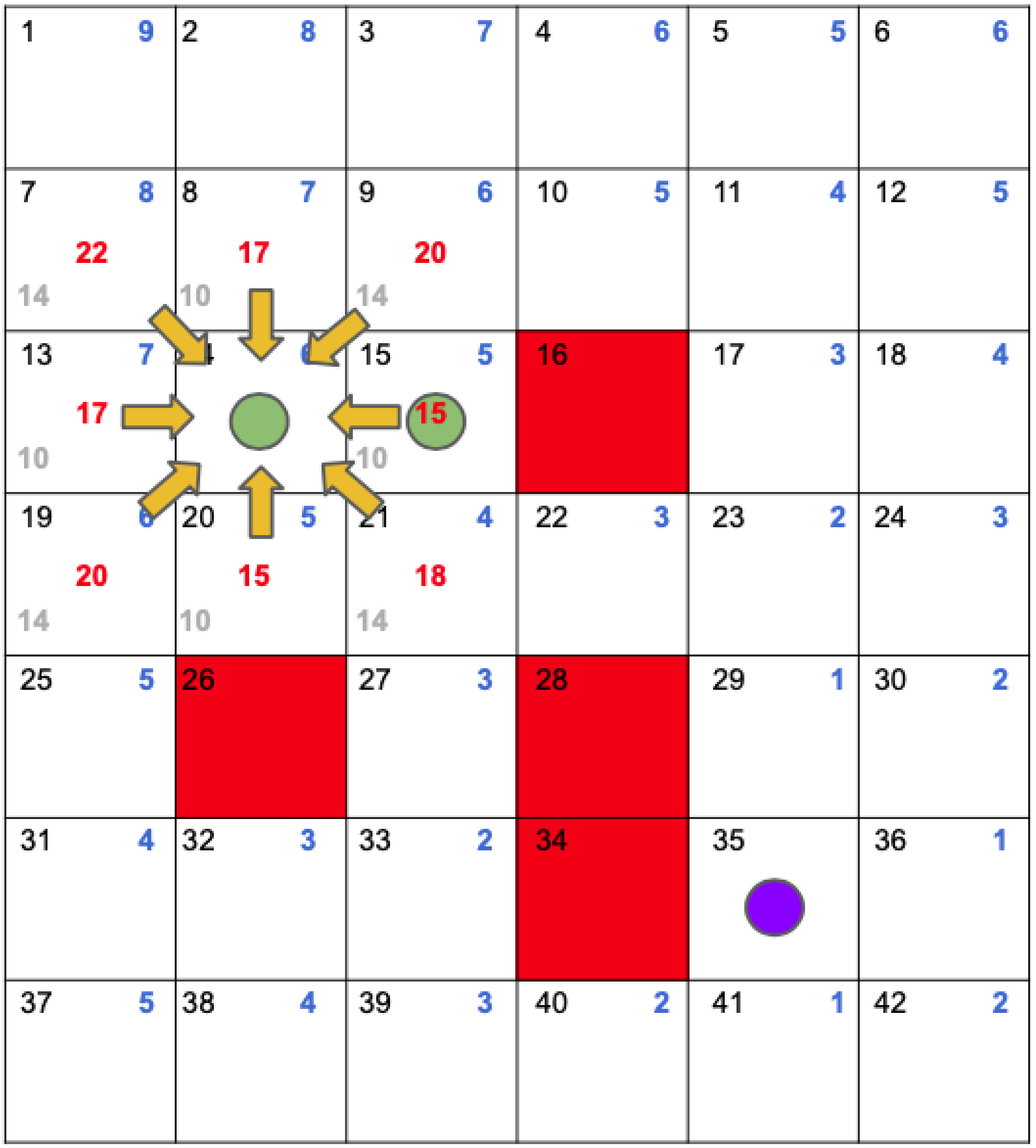
7,8,9,13,15,19,20,21

**Closed List**

14

**Now select the smallest F value**

# Pathfinding (A*)



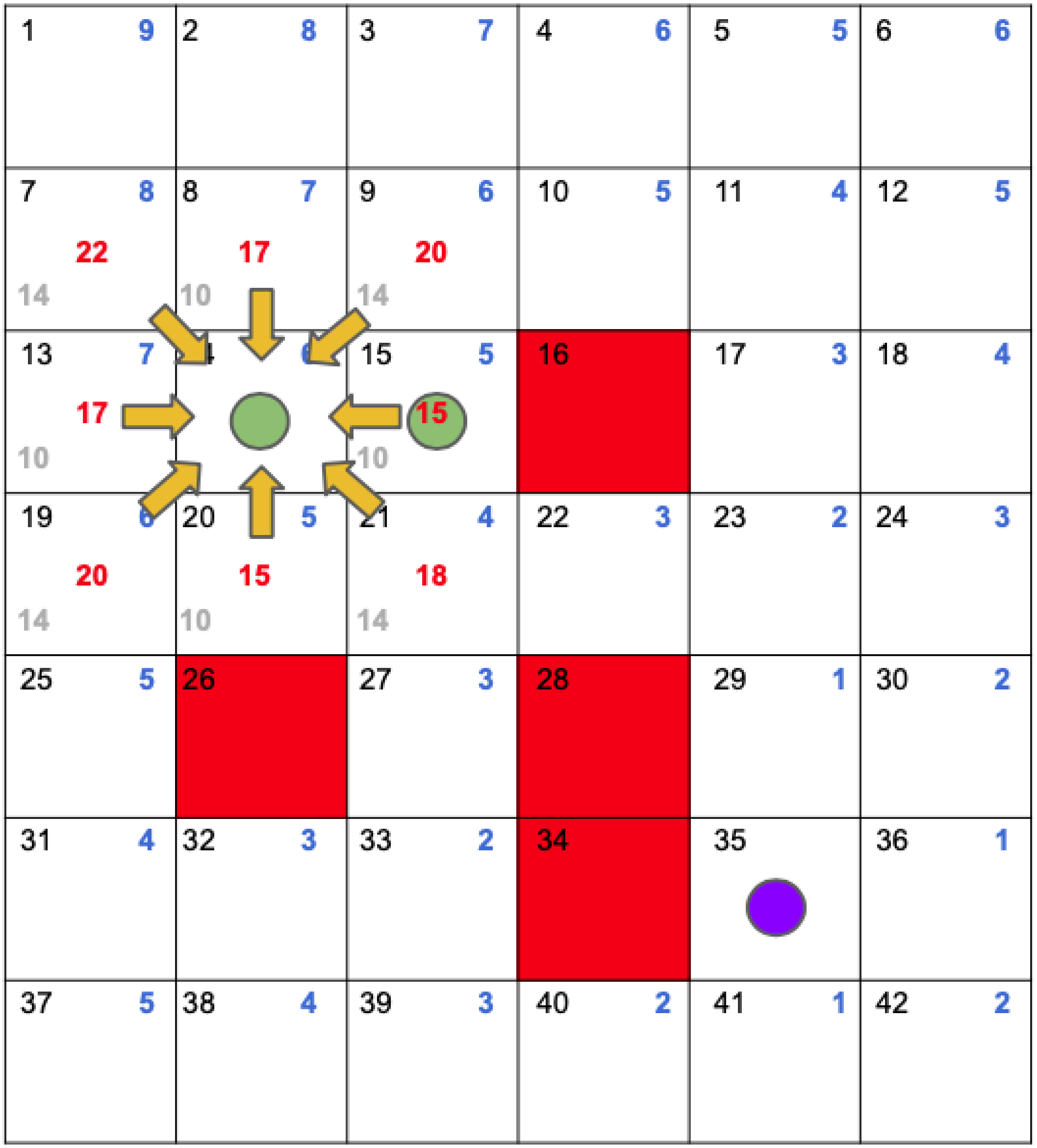
**Open List**

7,8,9,13,19,20,21

**Closed List**

14,15

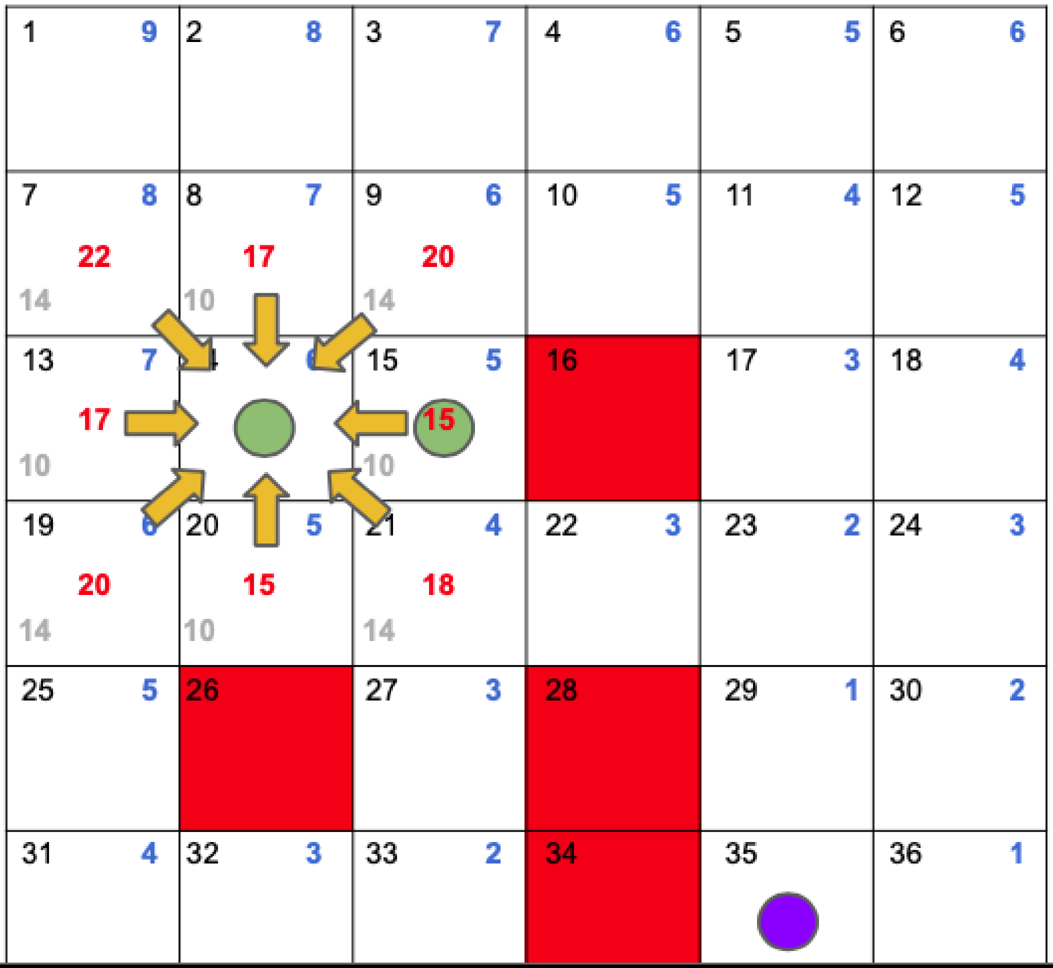
# Pathfinding (A*)



**Open List**
7,8,9,13,19,20,21,10,22

**Closed List**
14,15

# Pathfinding (A*)

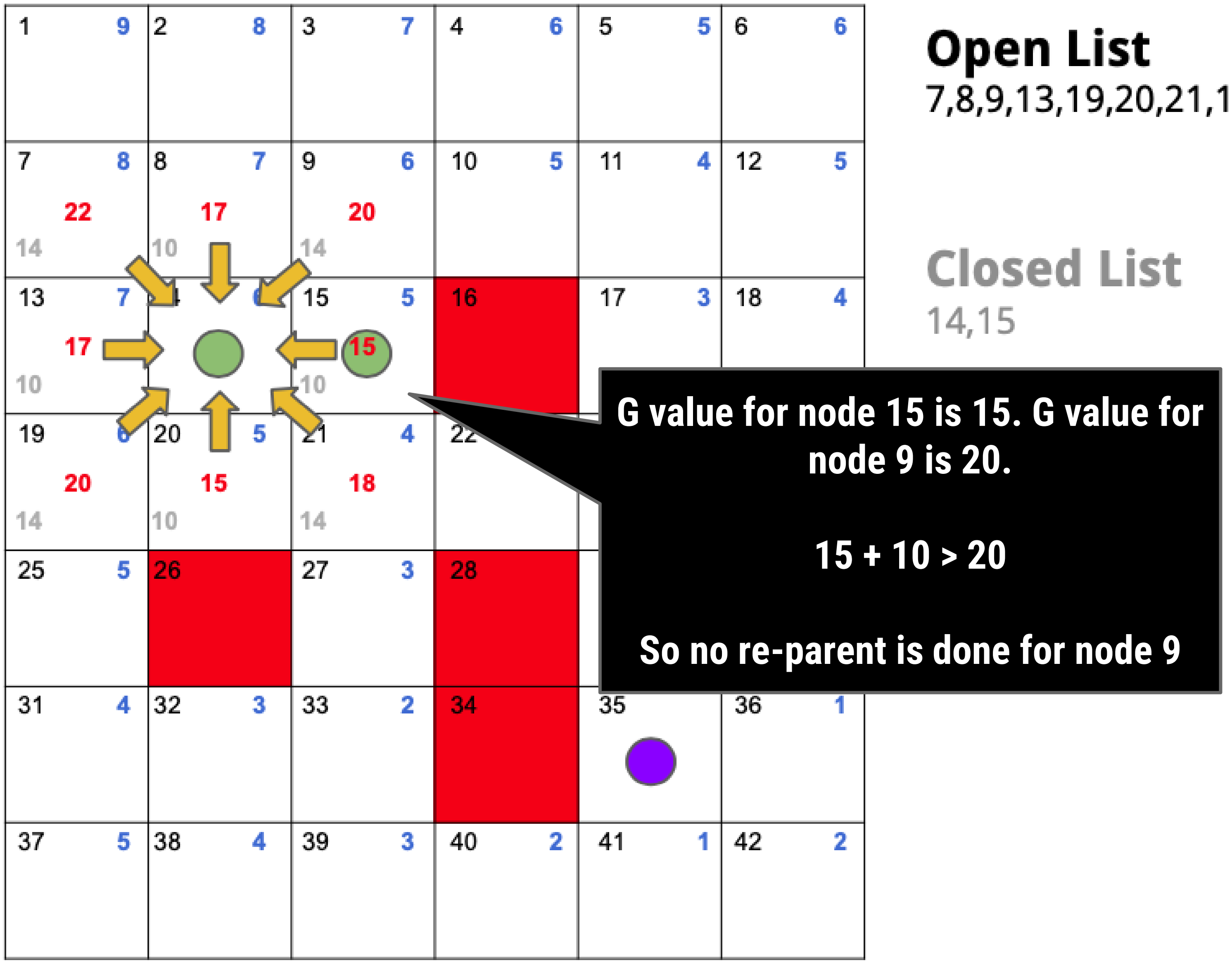


**Open List**
7,8,9,13,19,20,21,10,22

**Closed List**
14,15

**Re-parent nodes...**

# Pathfinding (A*)



G value for node 15 is 15. G value for node 9 is 20.

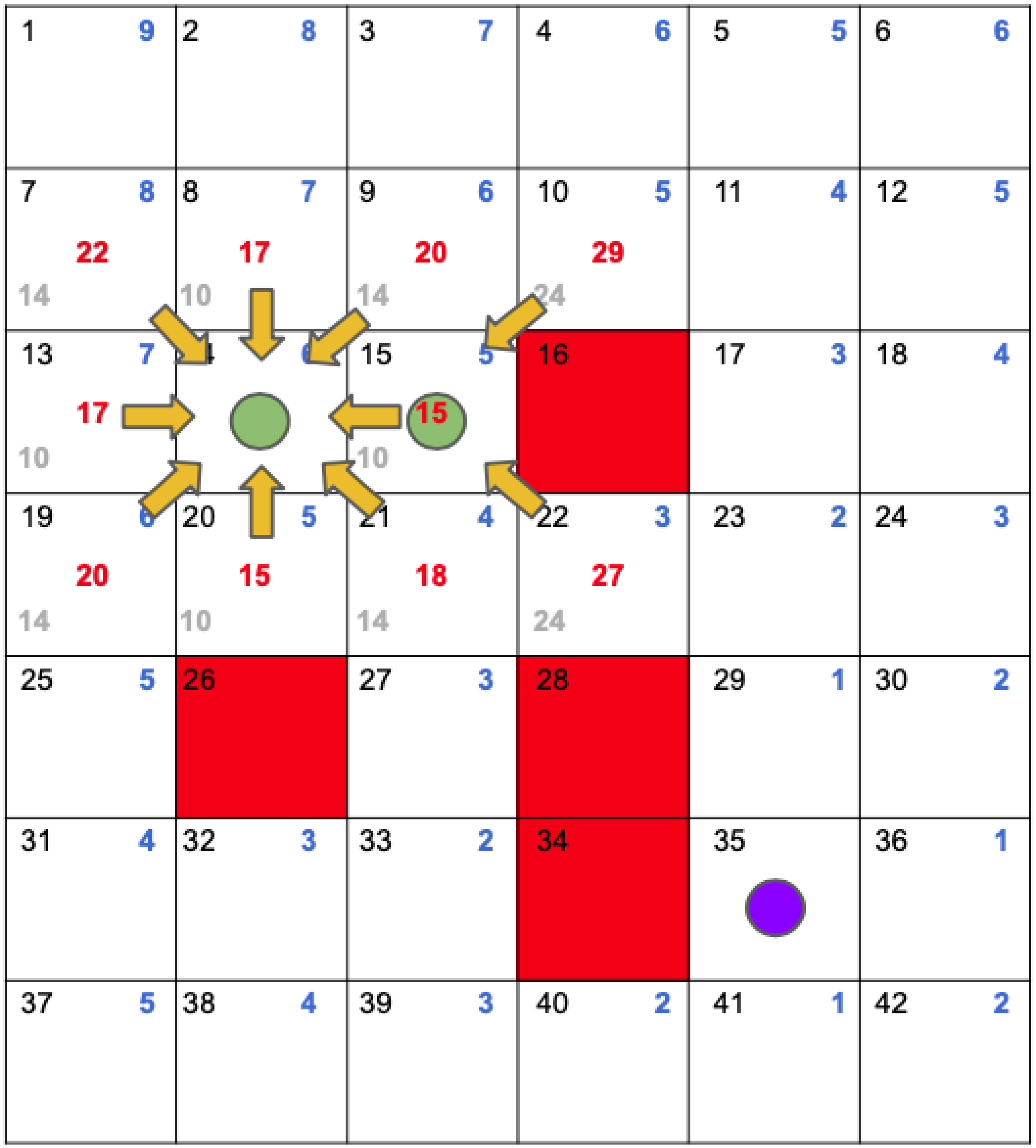
15 + 10 > 20

So no re-parent is done for node 9

**Open List**
7,8,9,13,19,20,21,10,22

**Closed List**
14,15

# Pathfinding (A*)



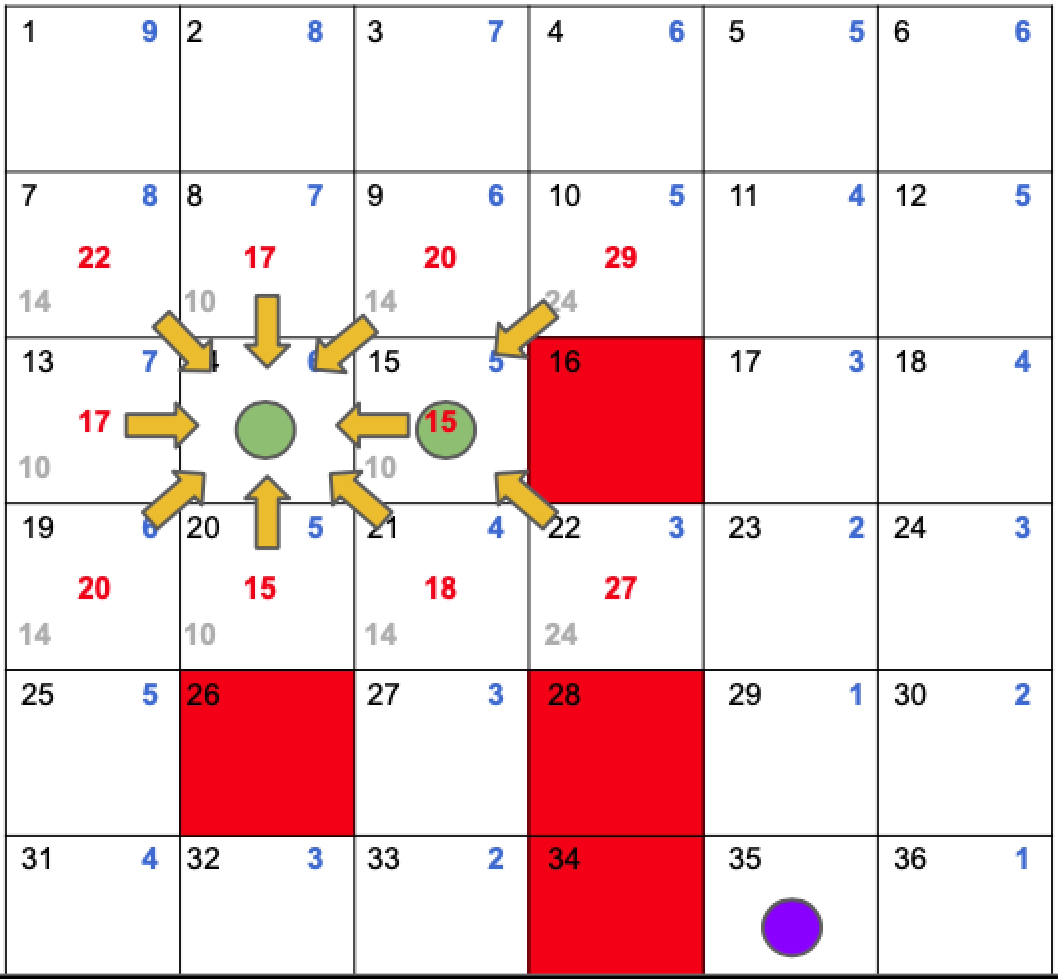
**Open List**

7,8,9,13,19,20,21,10,22

**Closed List**

14,15

## Pathfinding (A*)

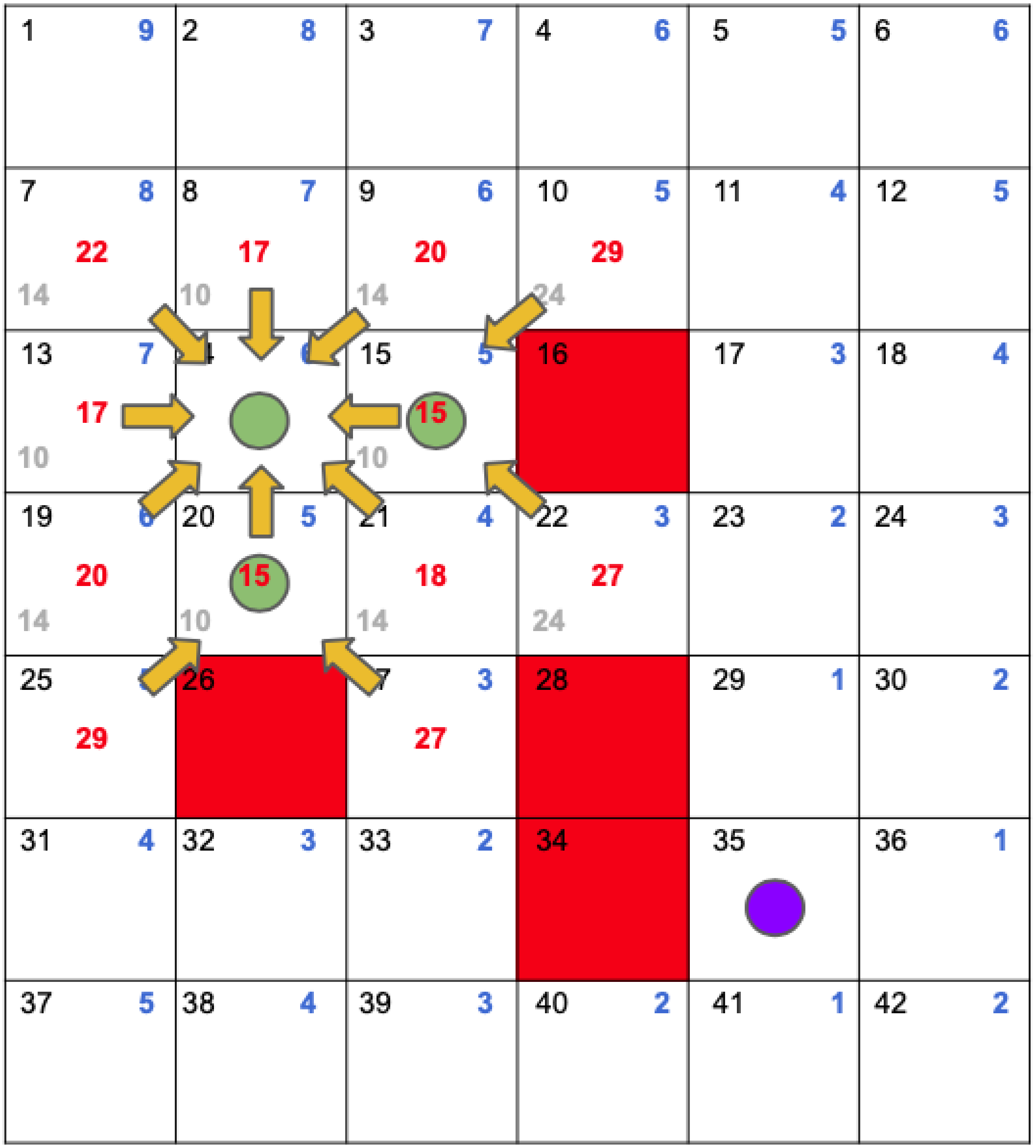


**Open List**
7,8,9,13,19,20,21,10,22

**Closed List**
14,15

**Repeat process...**

# Pathfinding (A*)



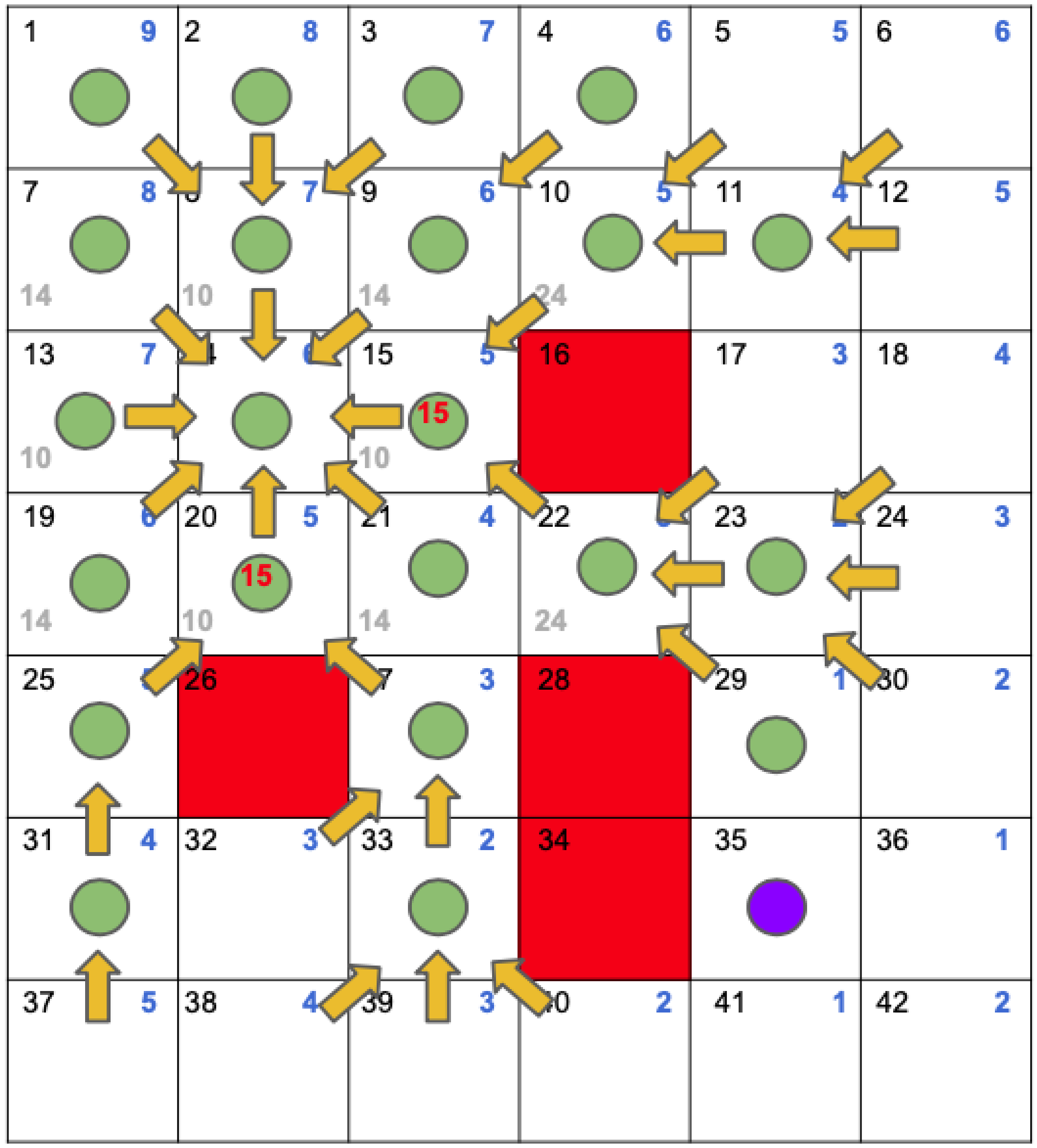
## Open List

7,8,9,13,19,21,10,22

## Closed List

14,15,20

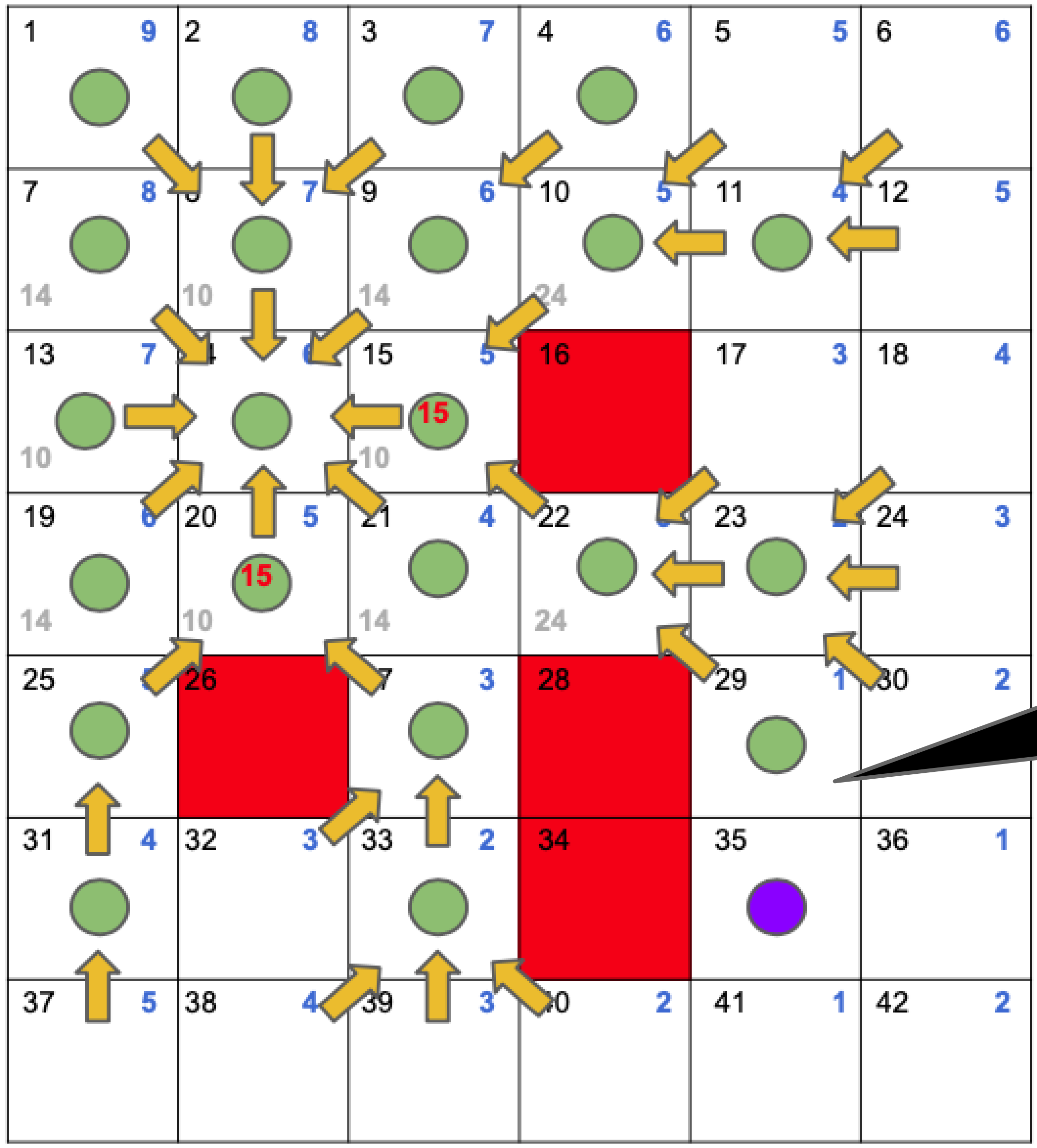
# Pathfinding (A*)



**Open List**
5,17,29,32,18,24,30,38,39,
40,6,12,37

**Closed List**
14,15,20,8,13,21,9,19,7,27,
2,10,25,22,3,1,4,23,33,31,2
9

# Pathfinding (A*)



**Open List**

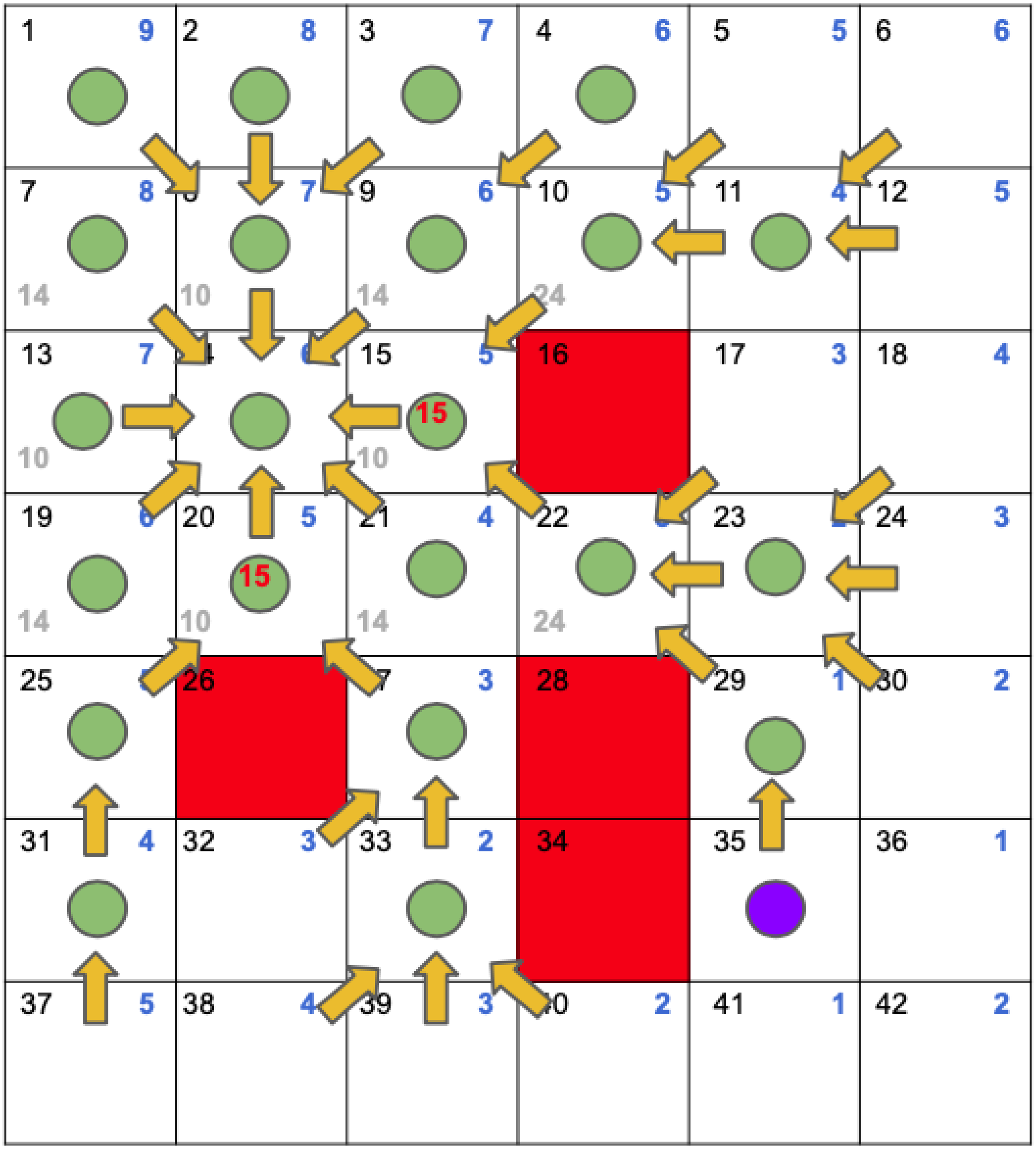
5,17,29,32,18,24,30,38,39,
40,6,12,37

**Closed List**

14,15,20,8,13,21,9,19,7,27,
2,10,25,22,3,1,4,23,33,31,2
9

**Stop! We are at only one position from the goal**

# Pathfinding (A*)



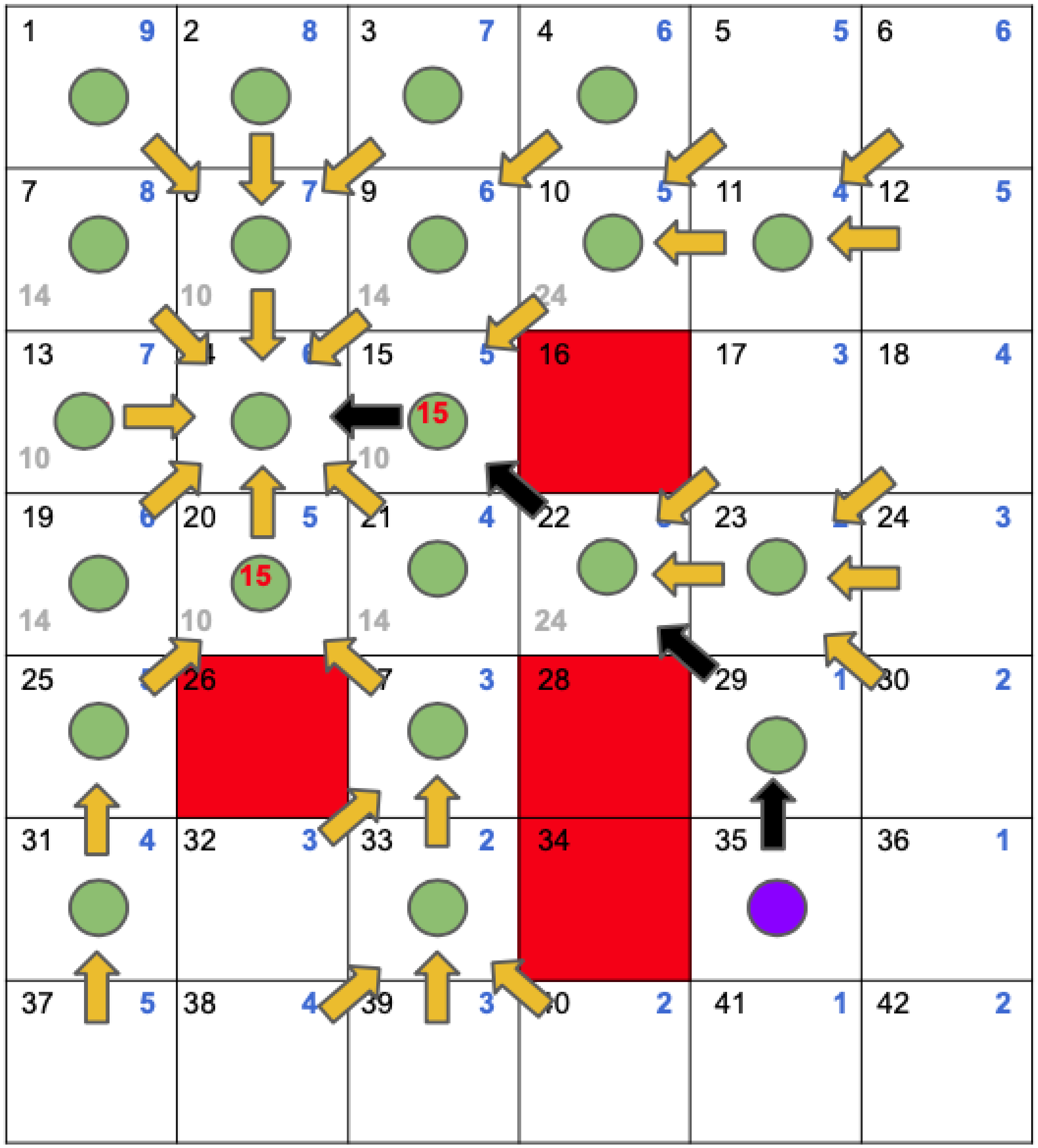
**Open List**

5,17,29,32,18,24,30,38,39,
40,6,12,37

**Closed List**

14,15,20,8,13,21,9,19,7,27,
2,10,25,22,3,1,4,23,33,31,2
9

# Pathfinding (A*)



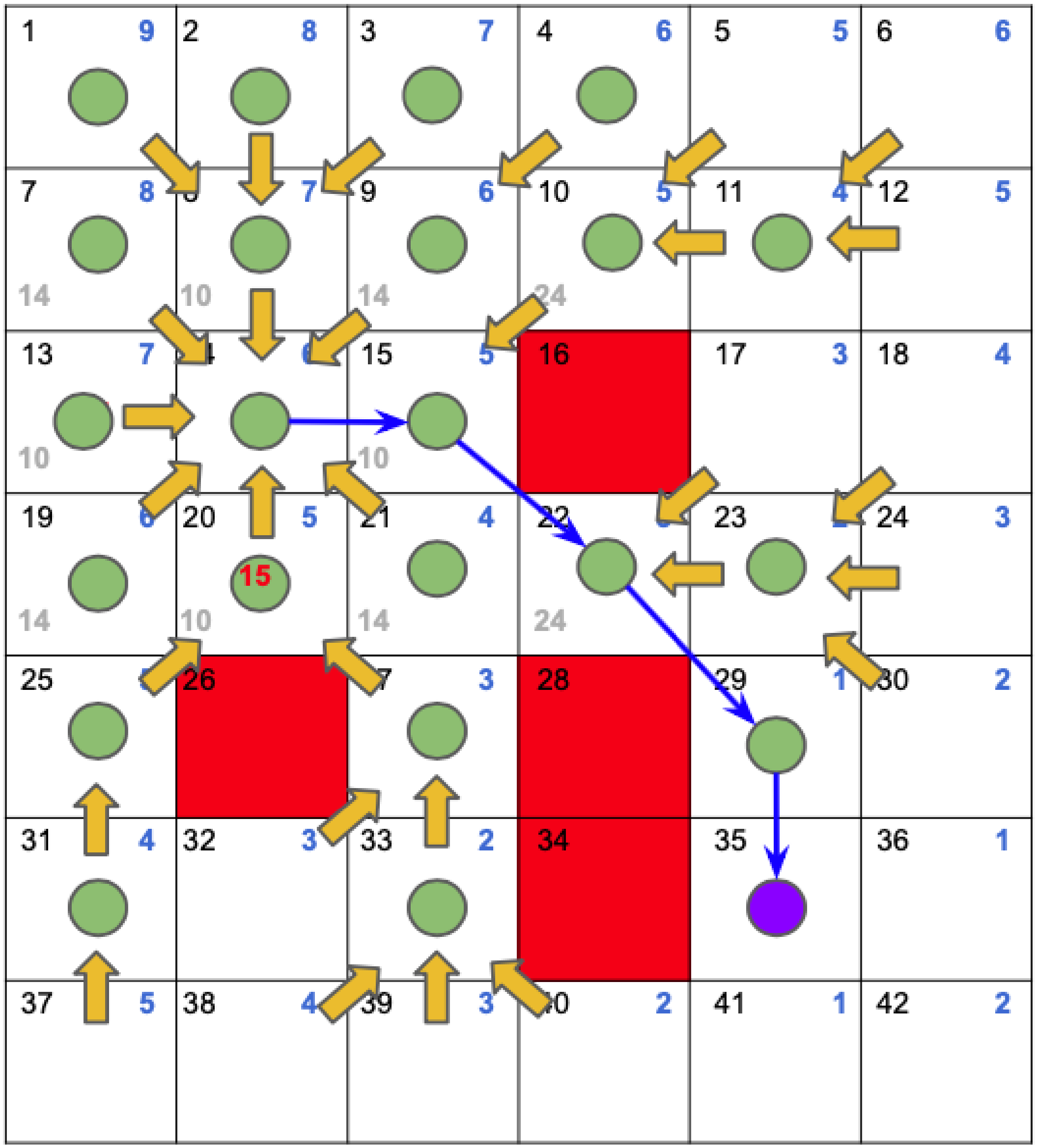
**Open List**

5,17,29,32,18,24,30,38,39,
40,6,12,37

**Closed List**

14,15,20,8,13,21,9,19,7,27,
2,10,25,22,3,1,4,23,33,31,2
9

# Pathfinding (A*)



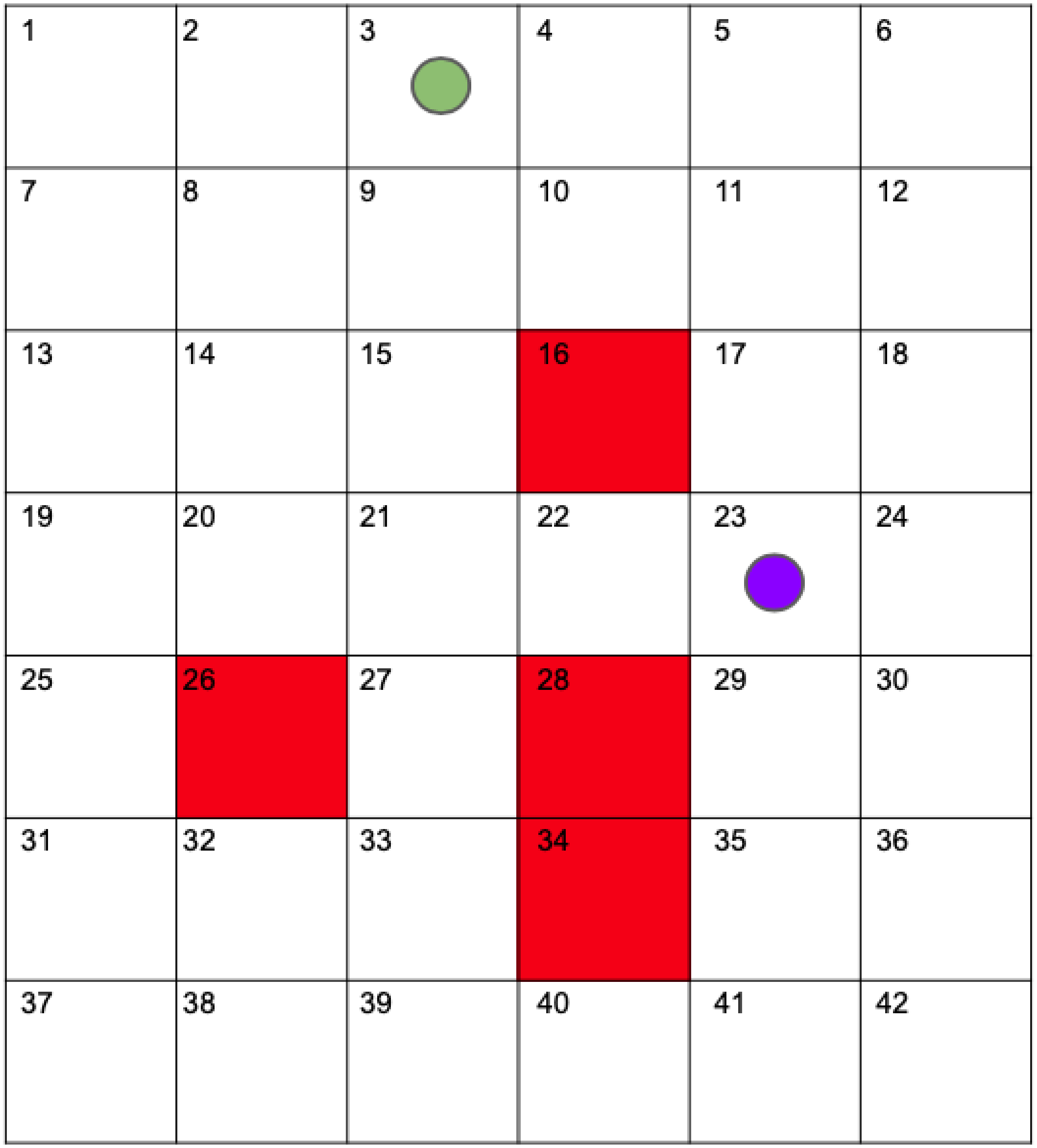
**Open List**

5,17,29,32,18,24,30,38,39,
40,6,12,37

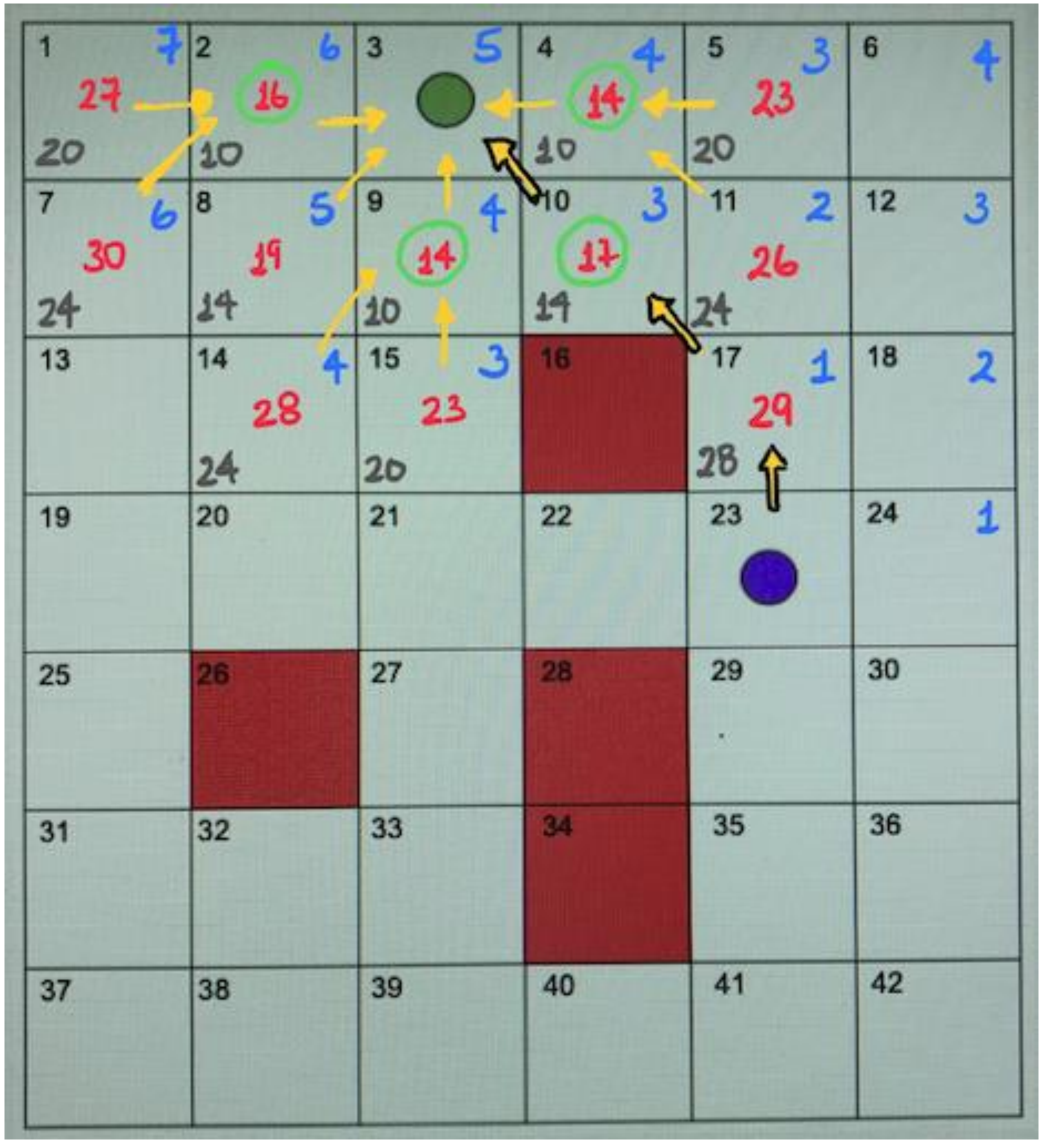
**Closed List**

14,15,20,8,13,21,9,19,7,27,
2,10,25,22,3,1,4,23,33,31,2
9

# Pathfinding (A*)



# Pathfinding (A*)



## Pathfinding (A*)

- There are variations of A* that make it faster or more adapted to certain circumstances:
  - **LPA*.** Similar to A* but can more quickly recalculate the best path when a small change to the graph is made
  - **D* Lite**. Based on LPA*, it does the same thing, but assumes the "start point" is a unit moving towards the finish while graph changes are being made
  - **HPA* (Hierarchical)**. Uses several layers at different abstraction levels to speed up the search. For instance, an higher level layer may simply connect rooms, while a lower level layer takes care of avoiding obstacles.
  - **IDA* (Iterative Deepening).** Reduces memory usage in comparison with regular A* by using iterative deepening.
  - **SMA* (Simplified Memory-Bounded).** Only makes use of available memory to carry out the search.
  - **Jump Point Search.** Speeds up pathfinding on uniform-cost grid maps.

# Search Algorithms

CE2103 - Algorithms and Data Structures II

## Let's practice!