COMP 4752
Computational Intelligence

Lecture 6

Heuristic Search / A*

Avoiding Repeated States

- One of the most important search ideas
- Most infinite loops / wasted time can be avoided by returning to identical states
- We must remember nodes visited
  - Don’t re-expand them
- Can lead to exponential savings in the number of nodes generated

Open and Closed Lists

- Closed List stores expanded nodes
- Open List = fringe of unexpanded nodes
- Often implemented as a hash table

General Graph-Search

```
1.  Function Graph-Search(problem)
2.      closed = empty set
3.      open = {Node(problem.initial_state)}
4.      while (true)
5.          if (open.empty) return fail
6.          node = remove_first(open)
7.          if (node.state is goal) return solution
8.          if (node.state not in closed)
9.              closed.add(node.state)
10.             open.add(Expand(node, problem))
```

Graph-Search Complexity

- Open and Closed lists store states
  - Can never re-visit a state once expanded (closed)
- In practice, time less than $O(b^d)$
  - Some problems still too large
- Graph-Search no longer optimal where Tree-Search may have been

Informed (Heuristic) Search

- An informed search strategy uses problem-specific knowledge beyond just the problem description itself
- Speeds up search times to goal
- Uses guesses (heuristics) to guide search toward the direction of the goal

Best-First Search (BeFS)

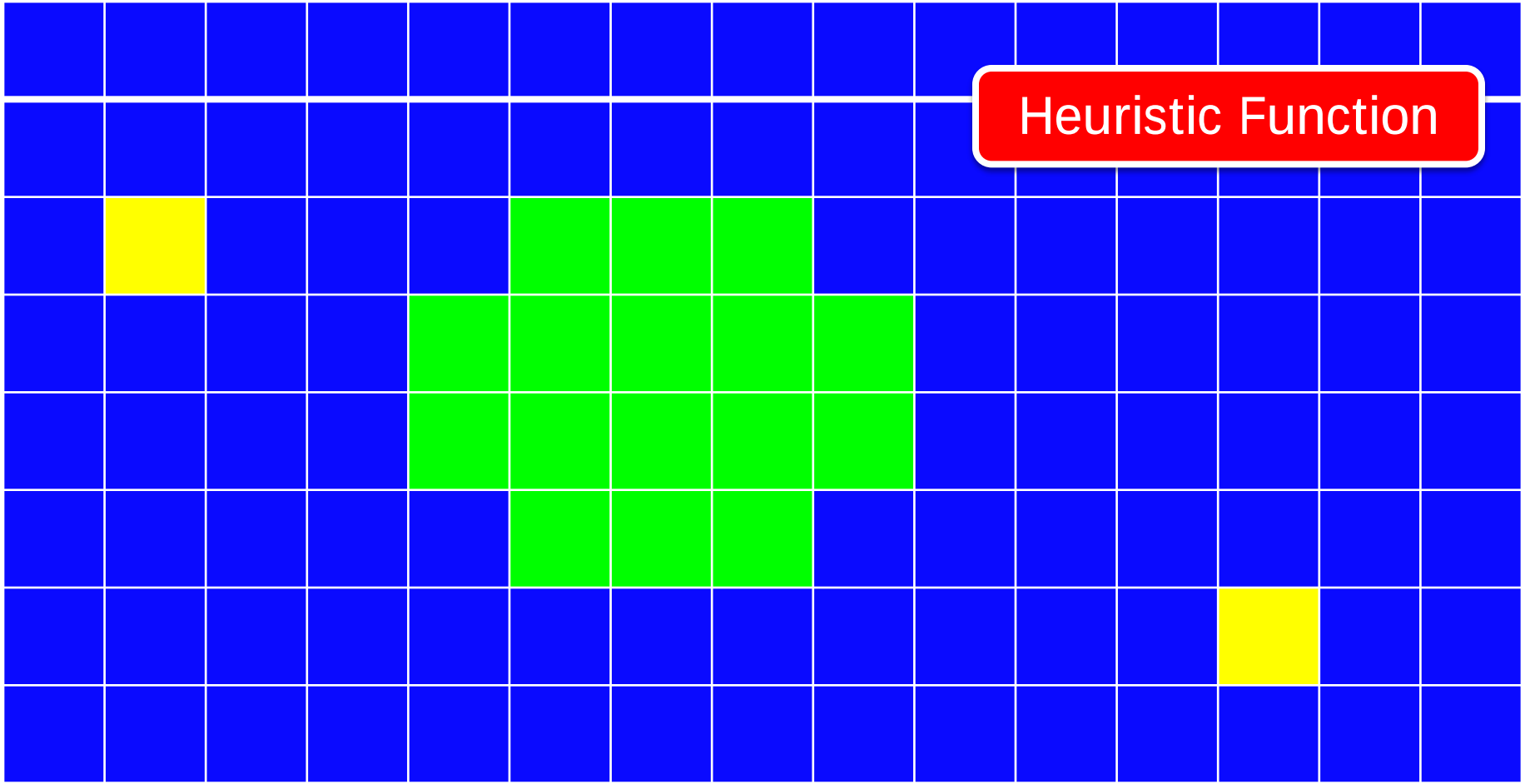
- Instance of general tree search
- Select a node for expansion based on an evaluation function f(n)
- Typically, select the node with minimum value of f(n), measures distance to goal
- “Best First” is slightly misleading
  - Knowing the true best node = straight to goal
  - More like “Best Guess First”

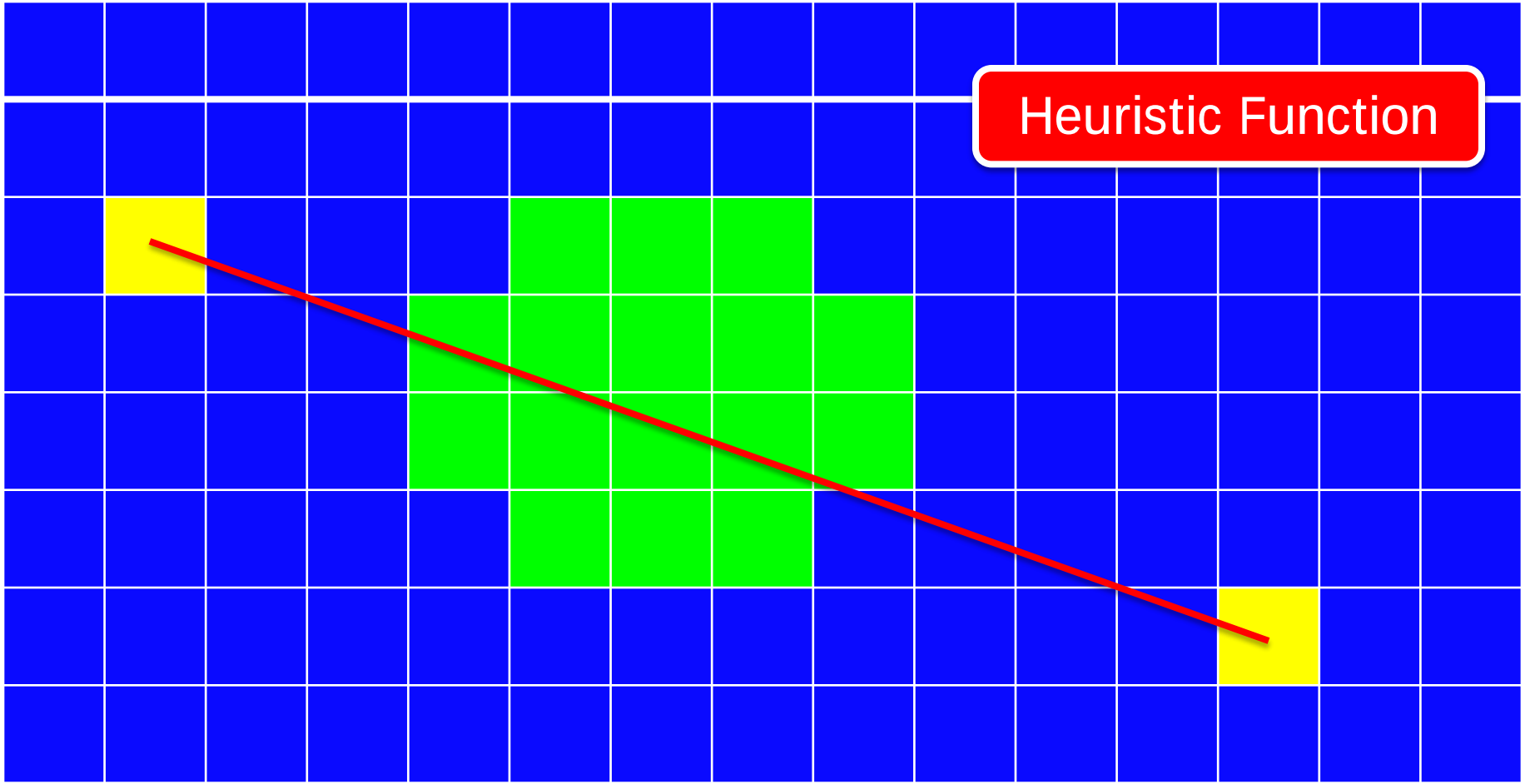
Best-First Search (BeFS)

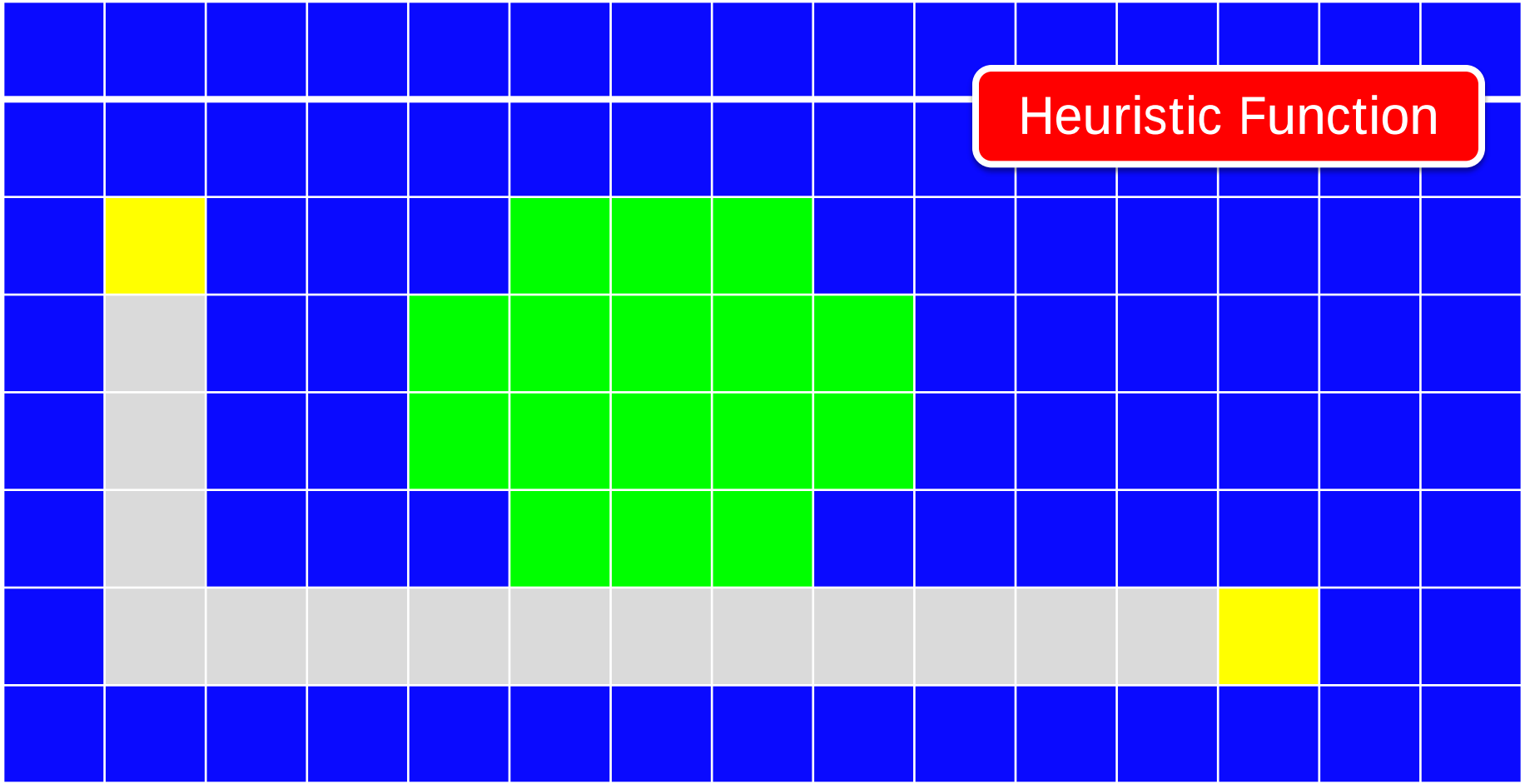
- BeFS can be implemented with general Tree-Search by using a priority queue for the open list, sorted on f(n)
- Whole family of BeFS algorithms
  - Have different evaluation functions
  - Most have a heuristic function h(n)

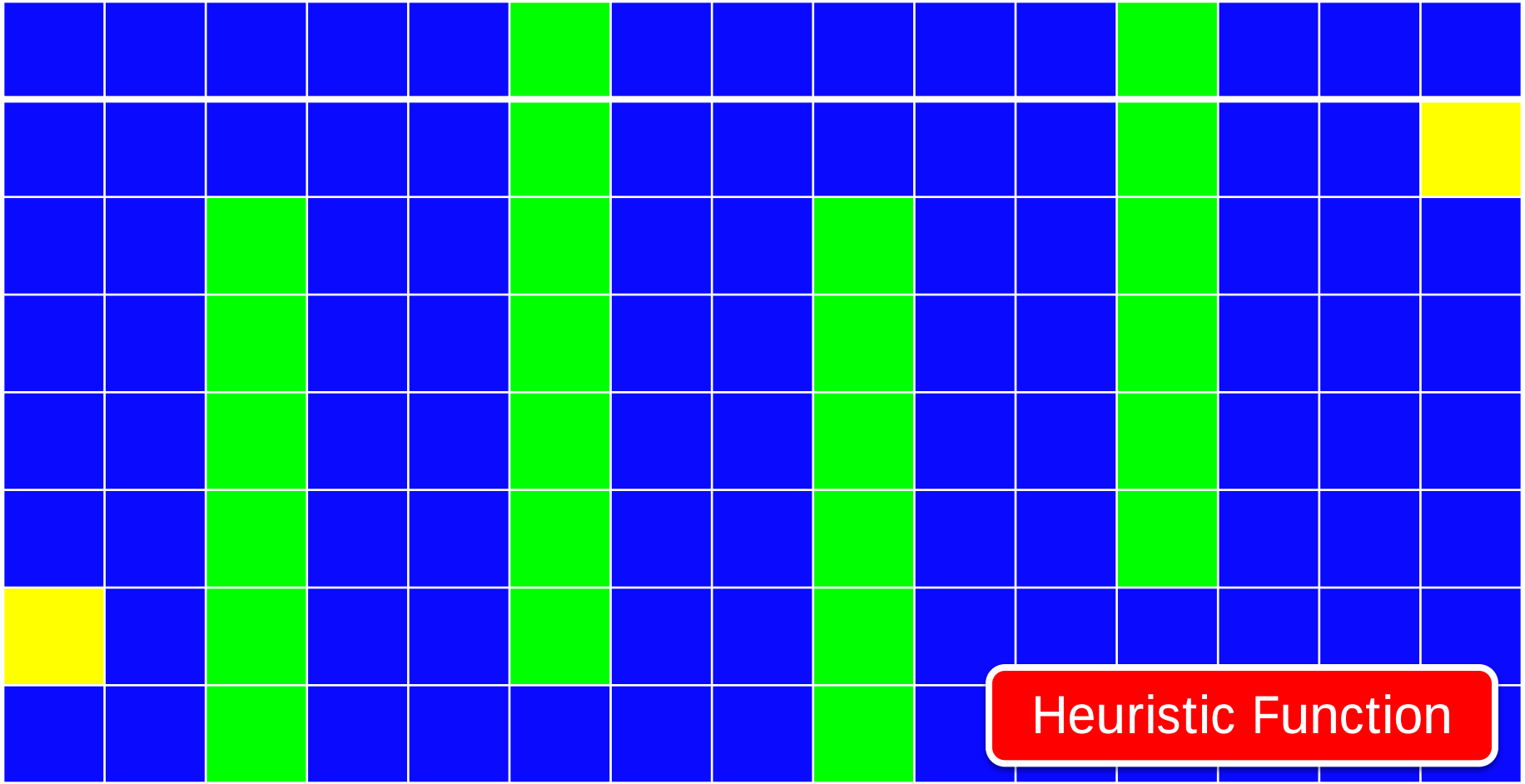
Heuristic Function h(n)

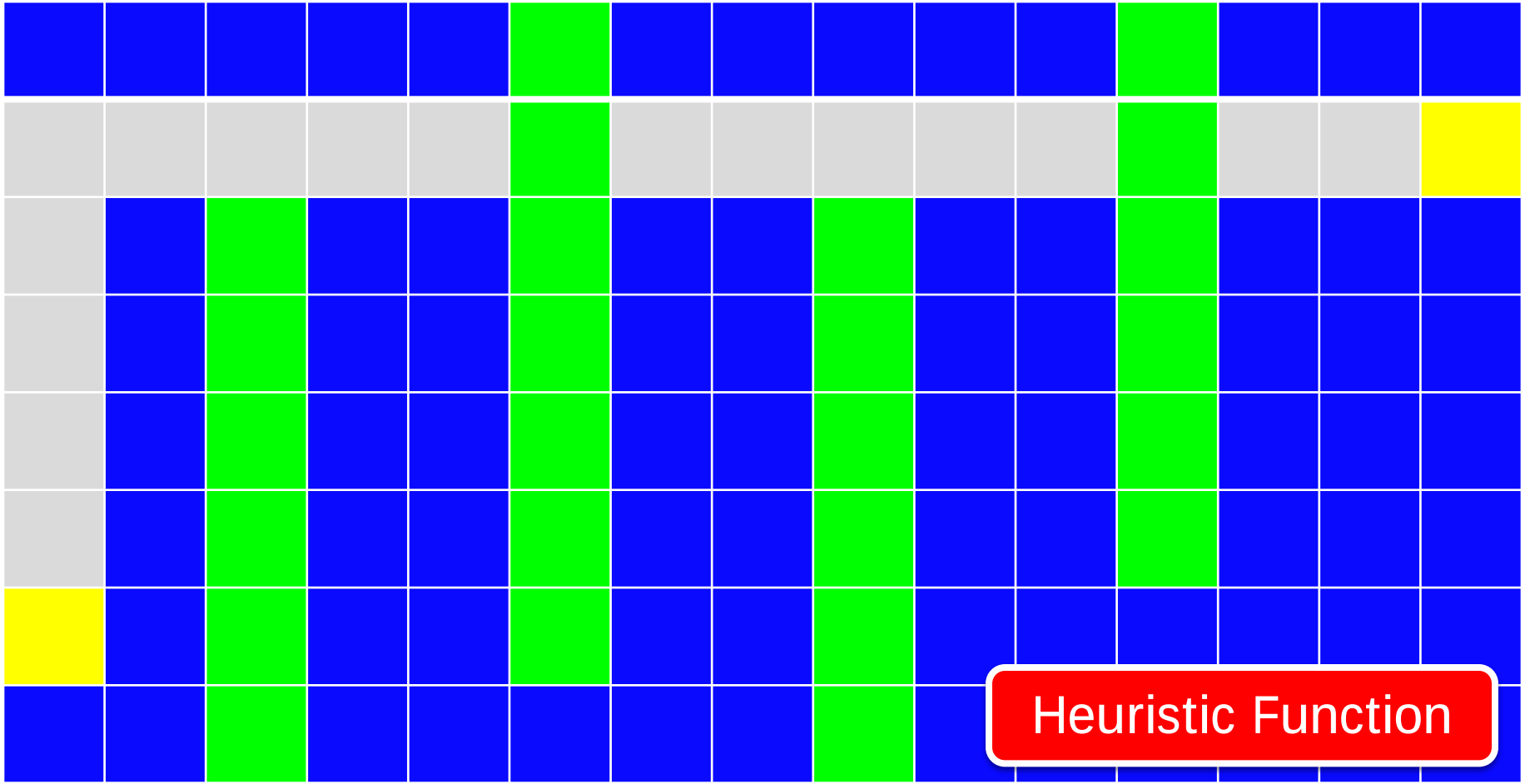
- Estimated cost of the optimal path from node n to a goal node
- Heuristic functions are the most common way that additional knowledge of a problem is given to the search algorithm
- If n is a goal node, h(n) = 0
- Perfect heuristic = h*(n)


Heuristic Function


Heuristic Function

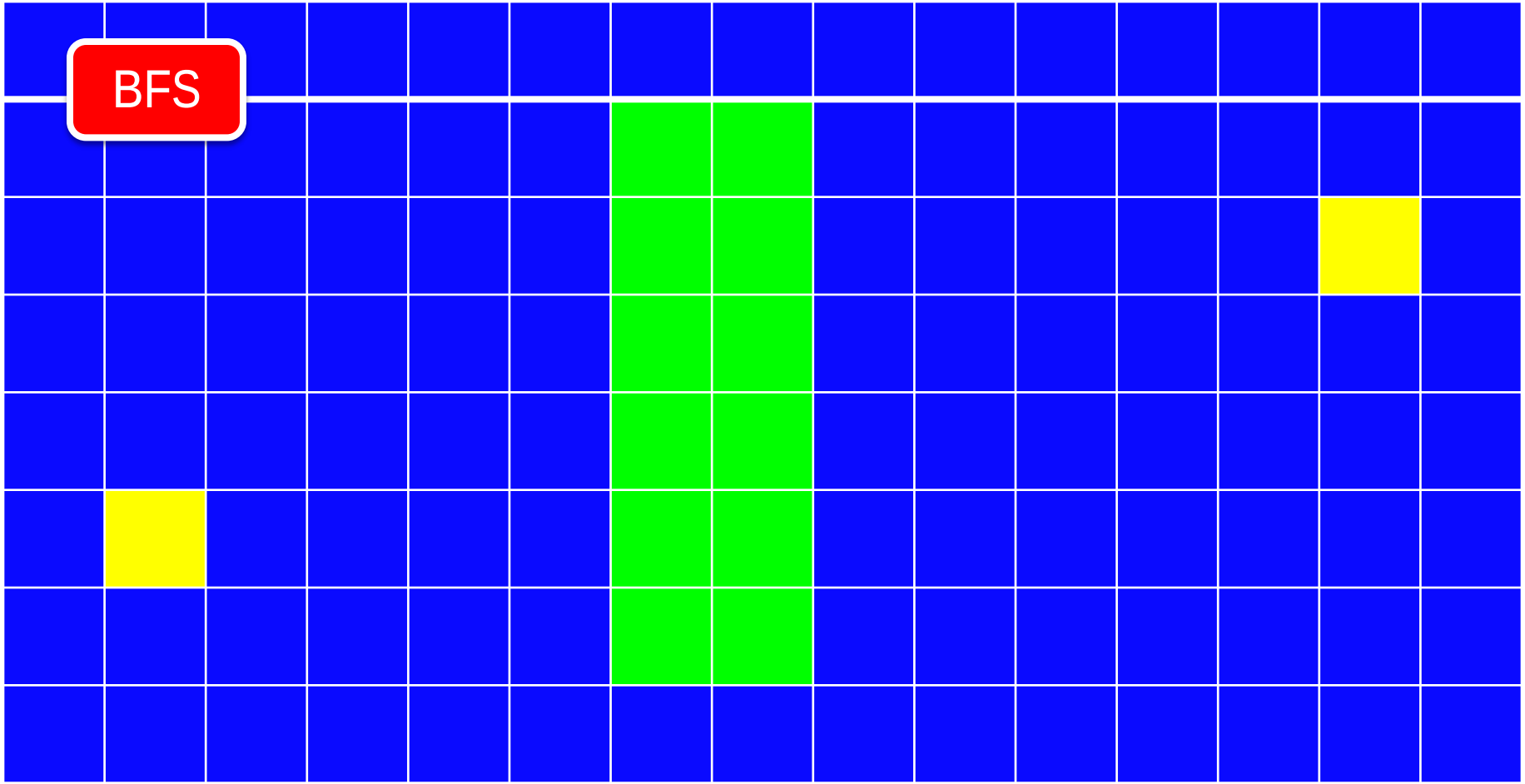

Heuristic Function

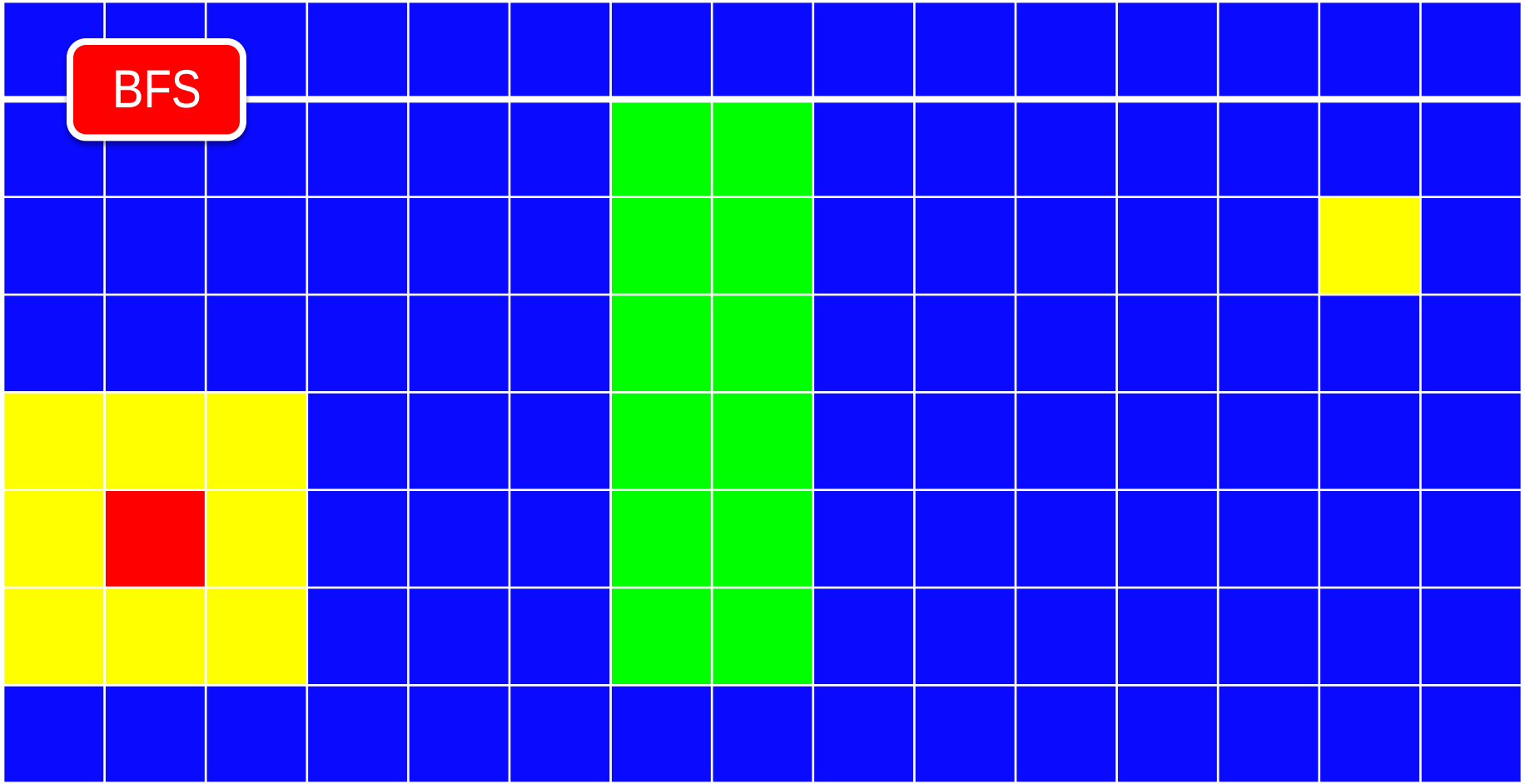

Heuristic Function

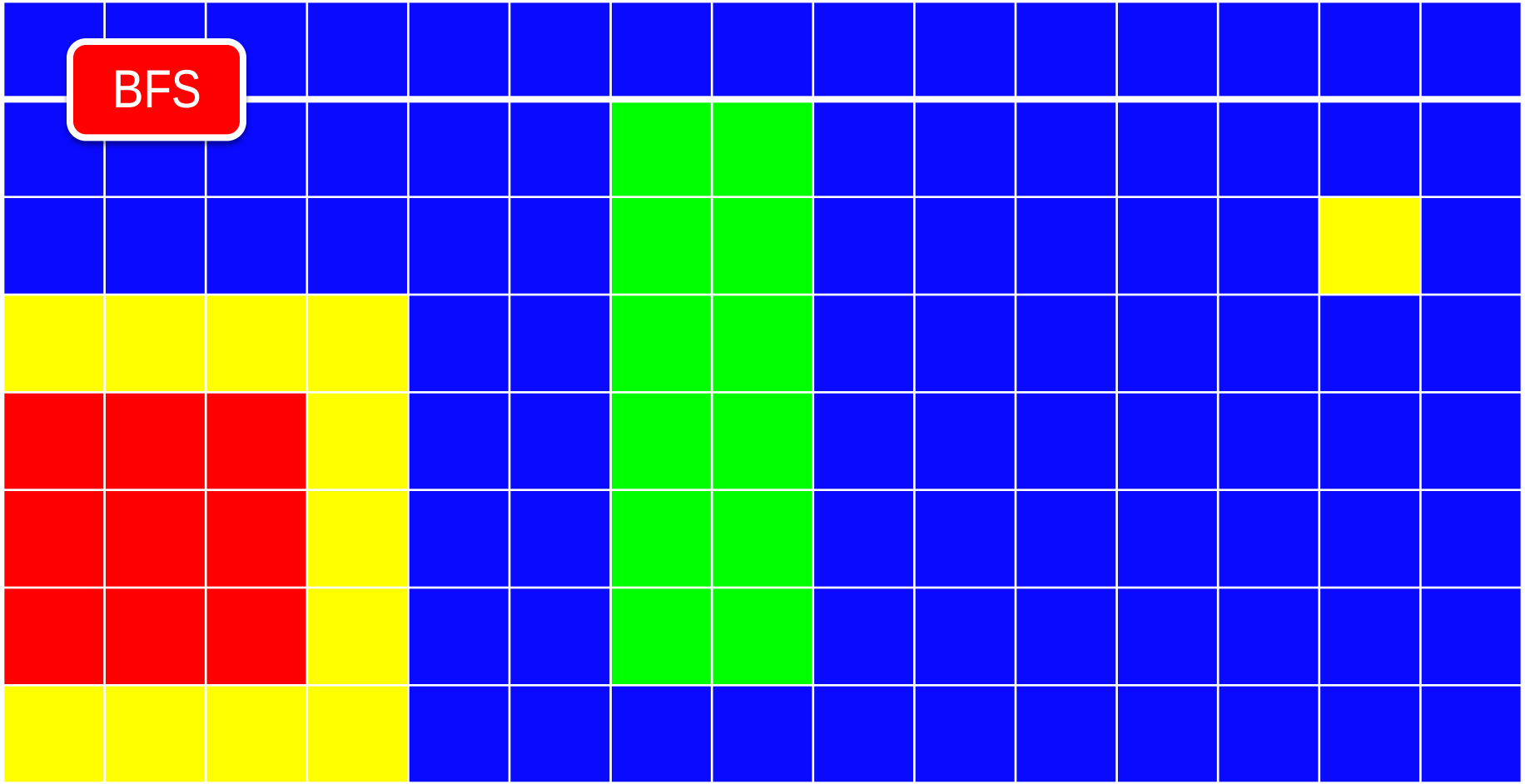

Heuristic Function

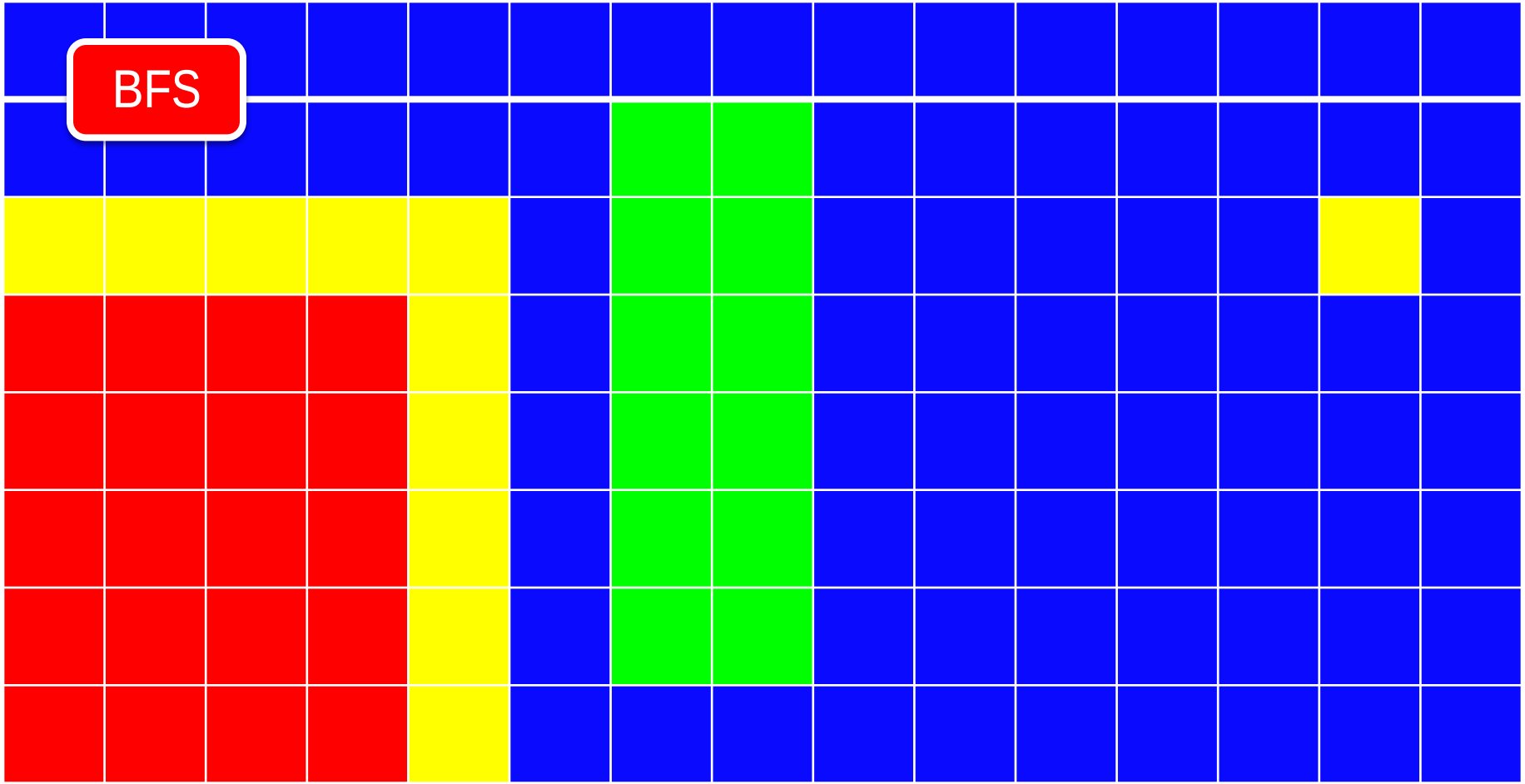
Greedy Best-First Search

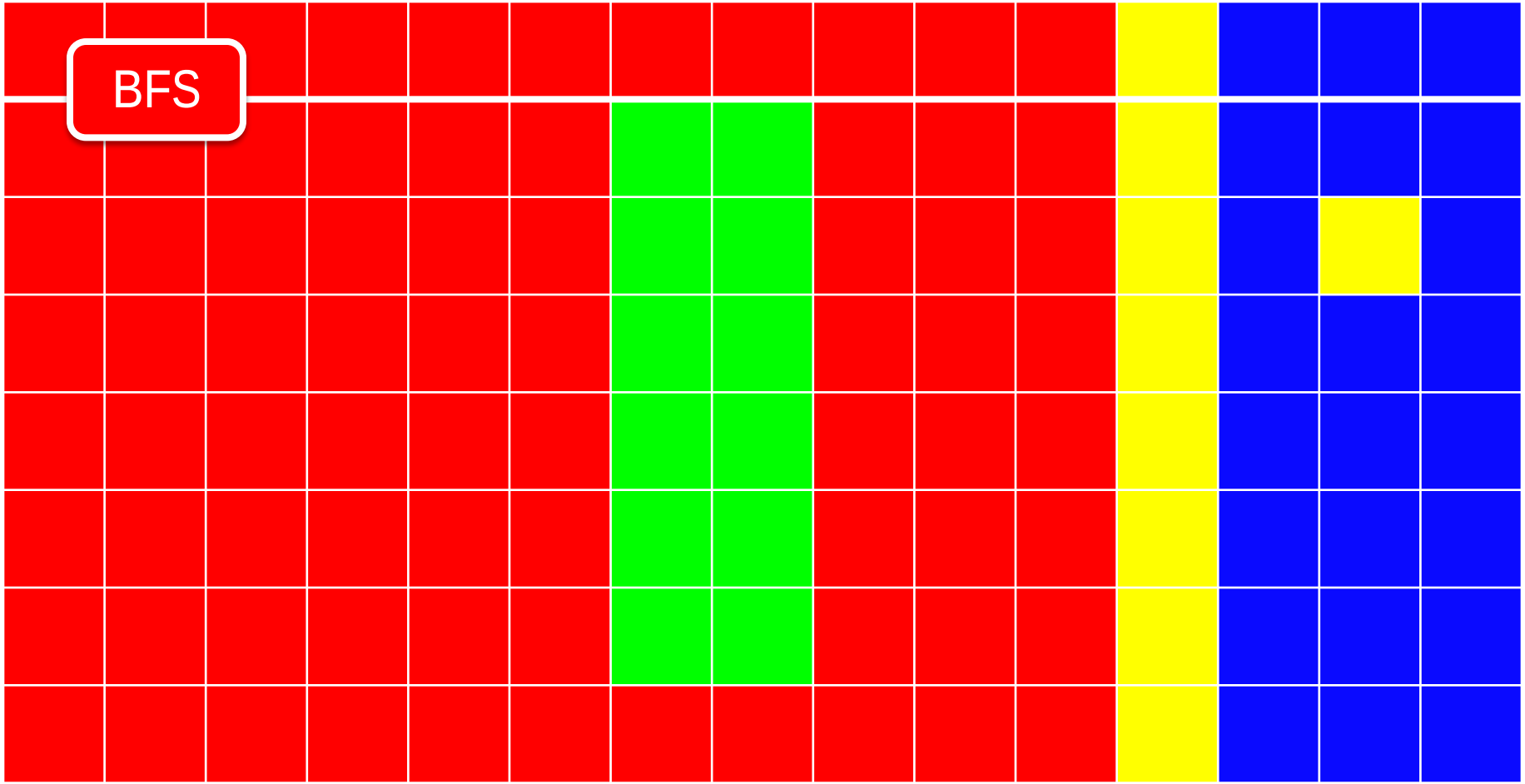
- Expands the node that it thinks is closest to the goal node
- This will hopefully lead us to the goal
- Thus, for GBeFS: f(n) = h(n)
- Resembles DFS
  - Tries a single path all the way to a goal
  - Backs up when it hits a dead-end

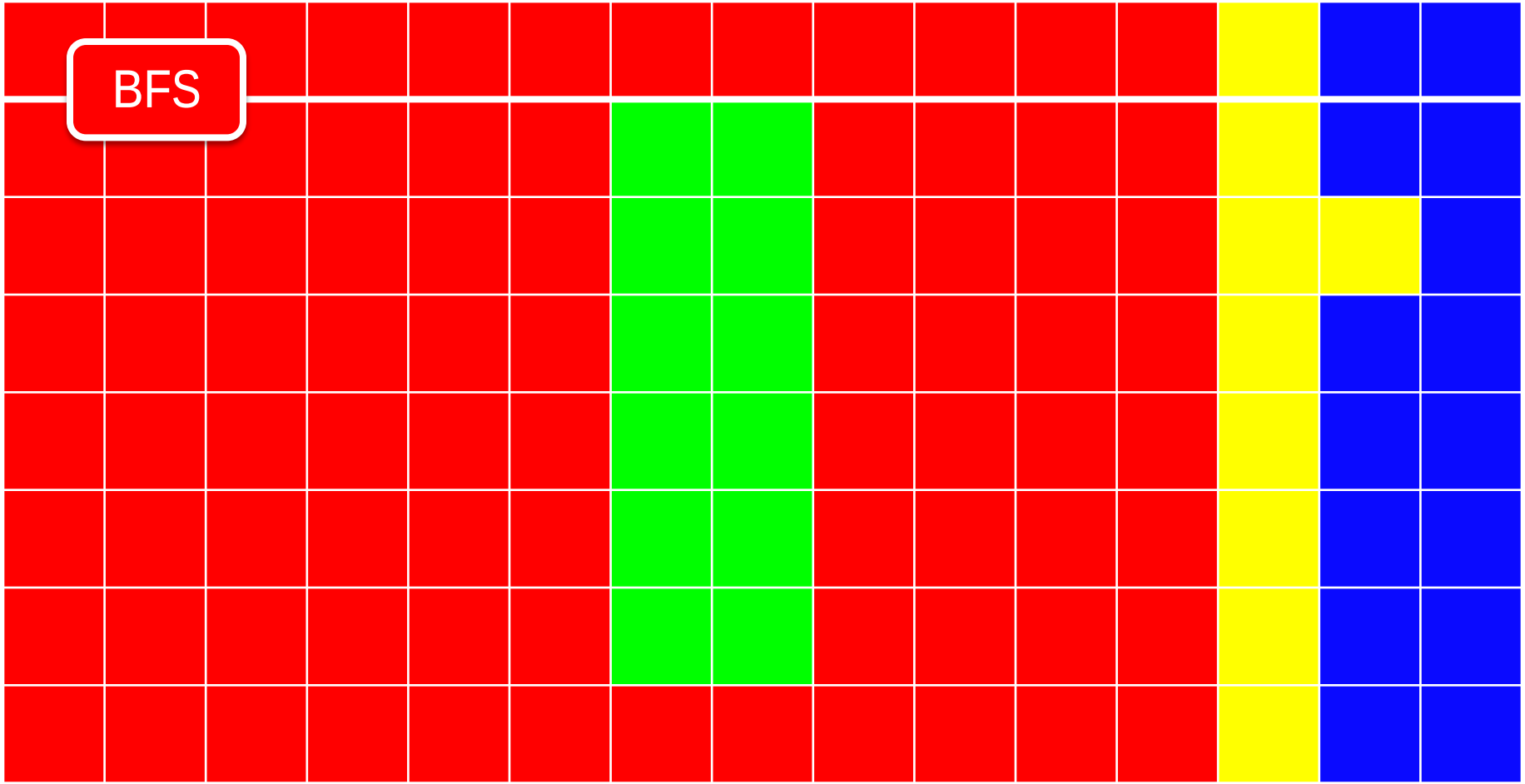

BFS

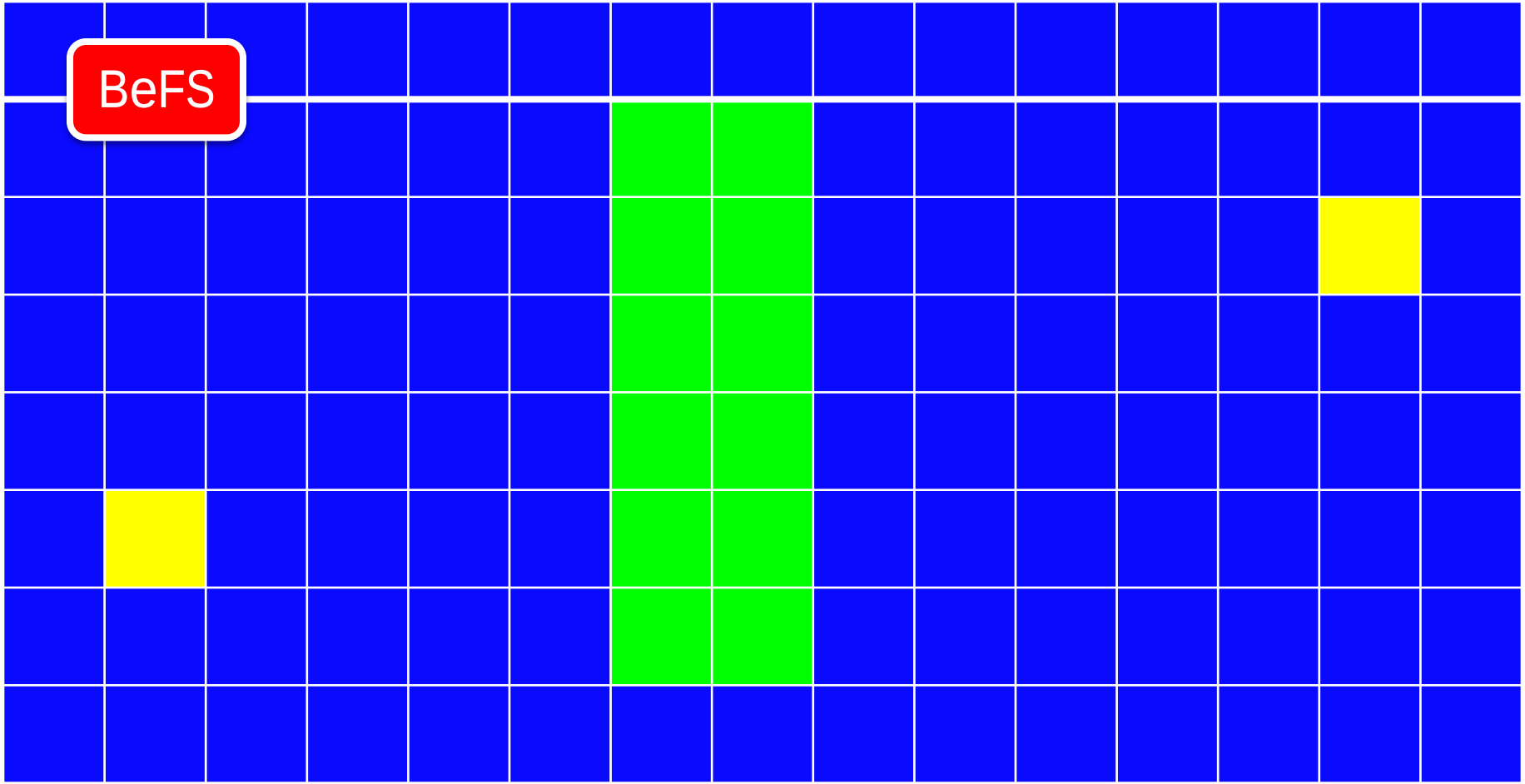

BFS

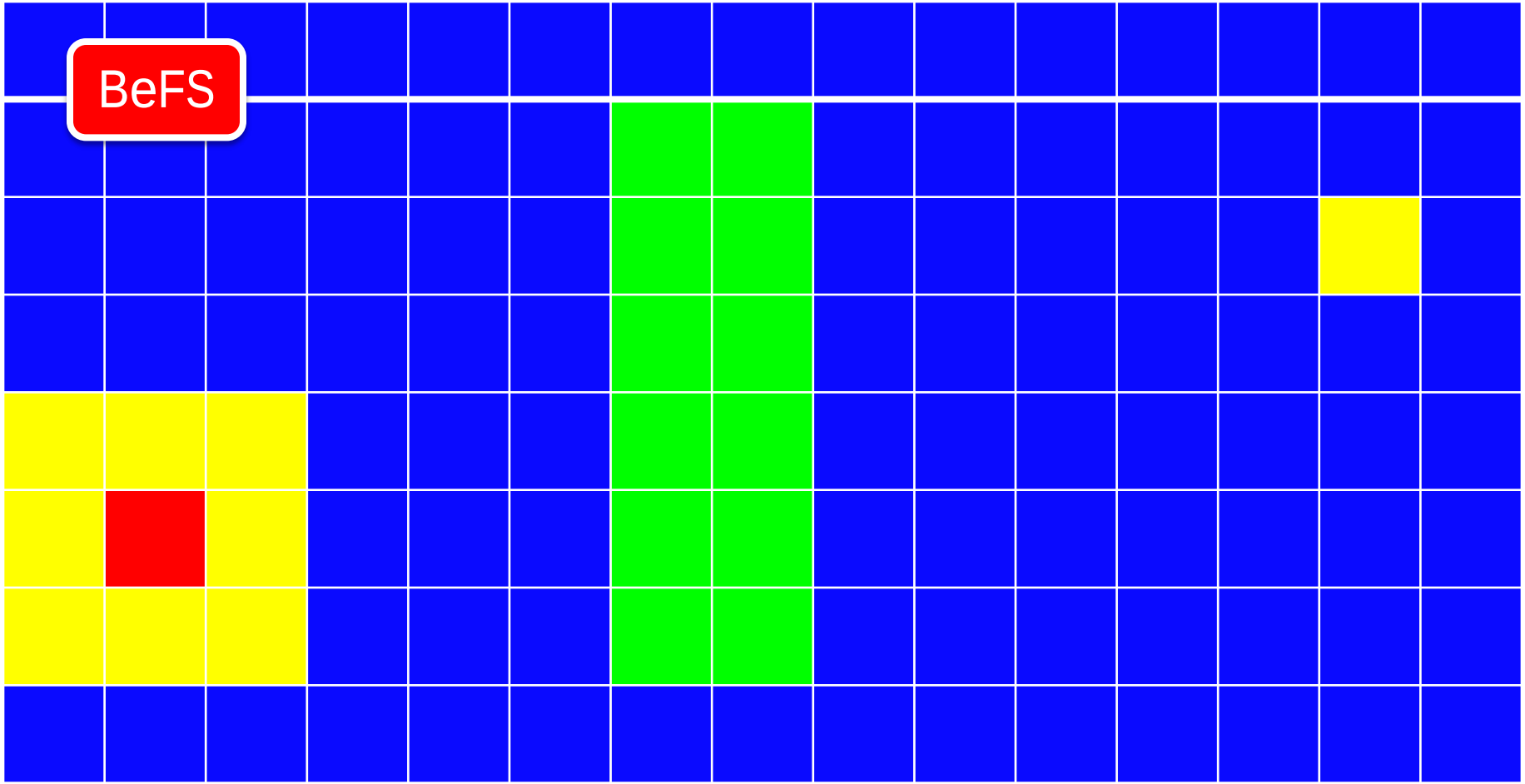

BFS

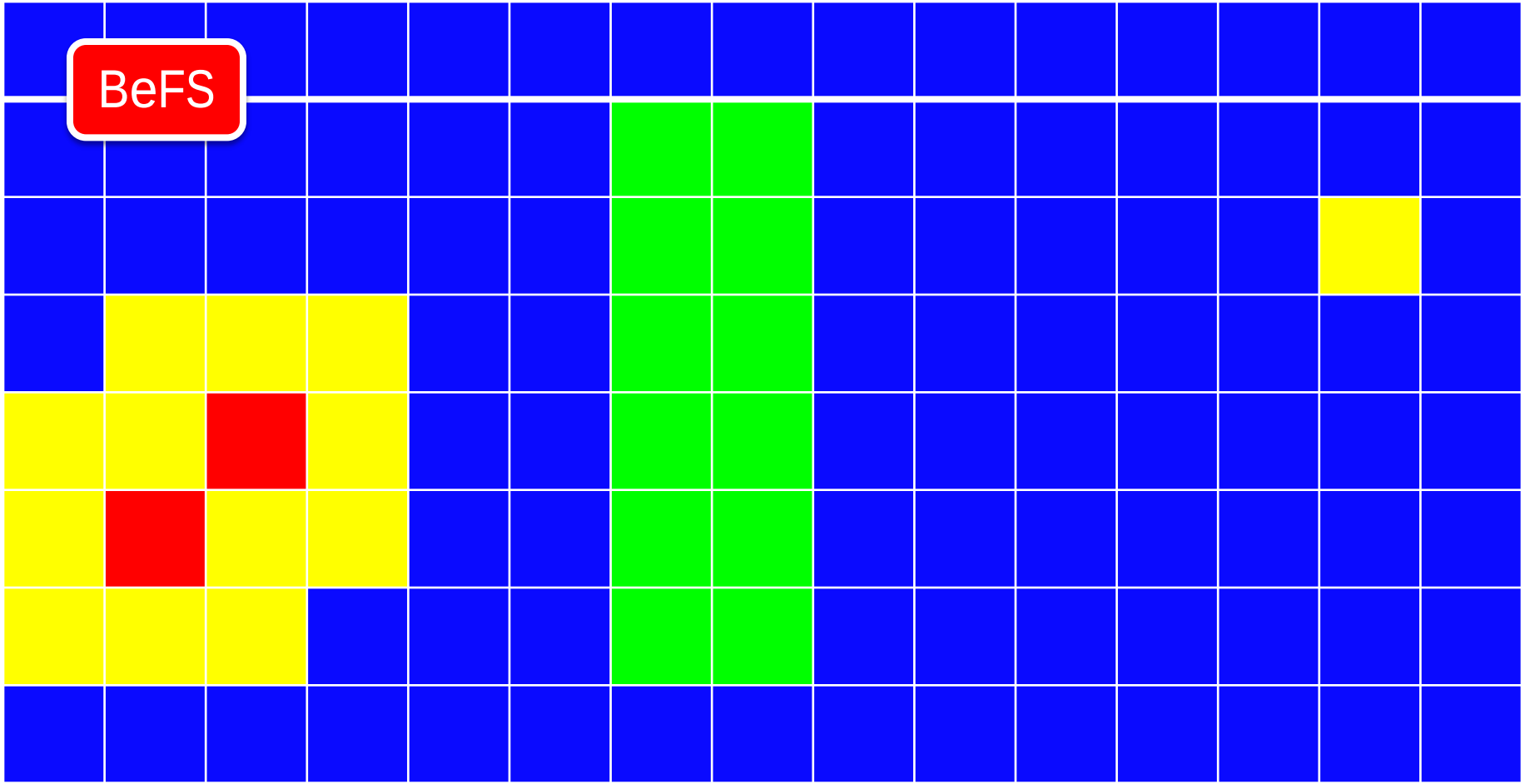

BFS

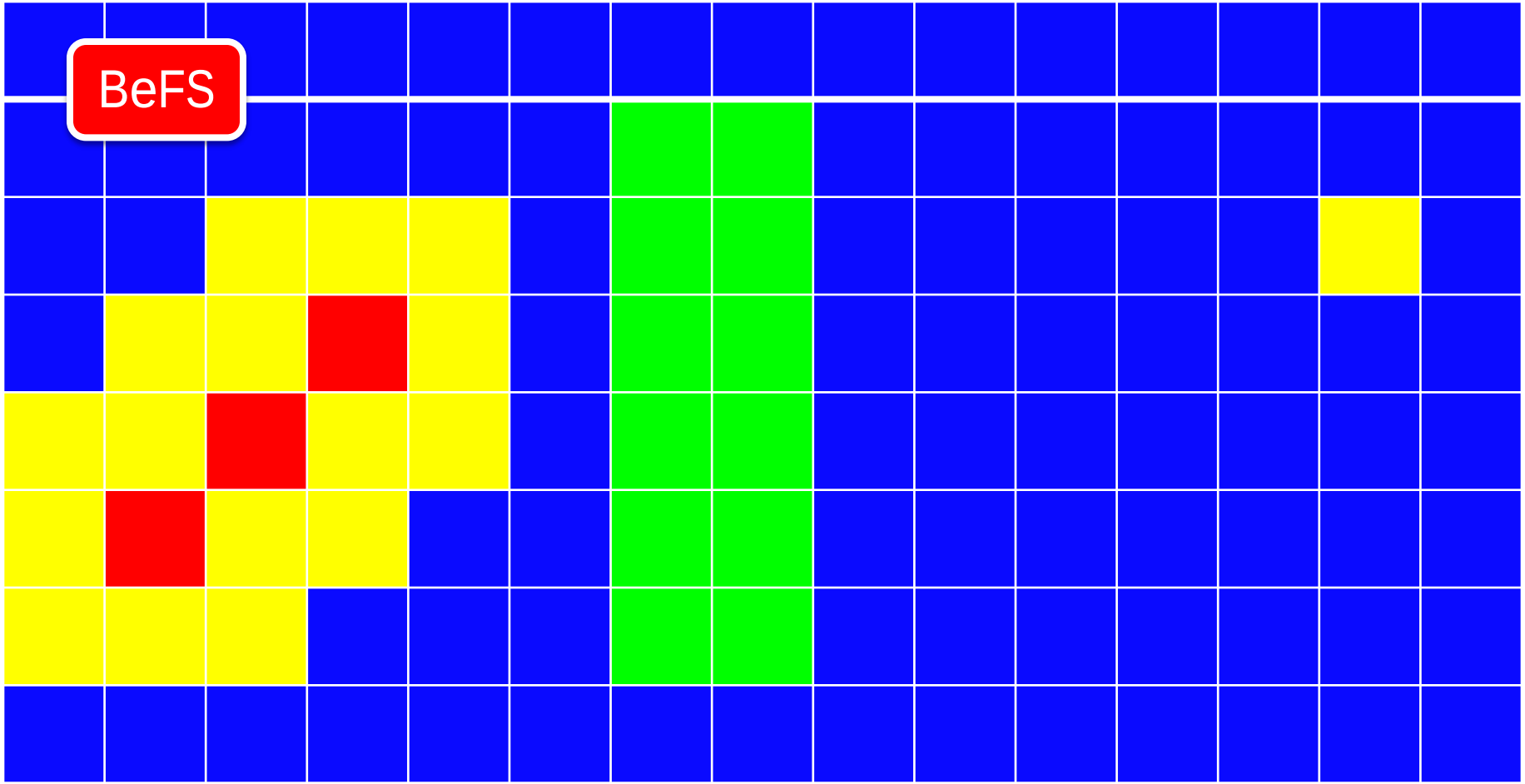

BFS

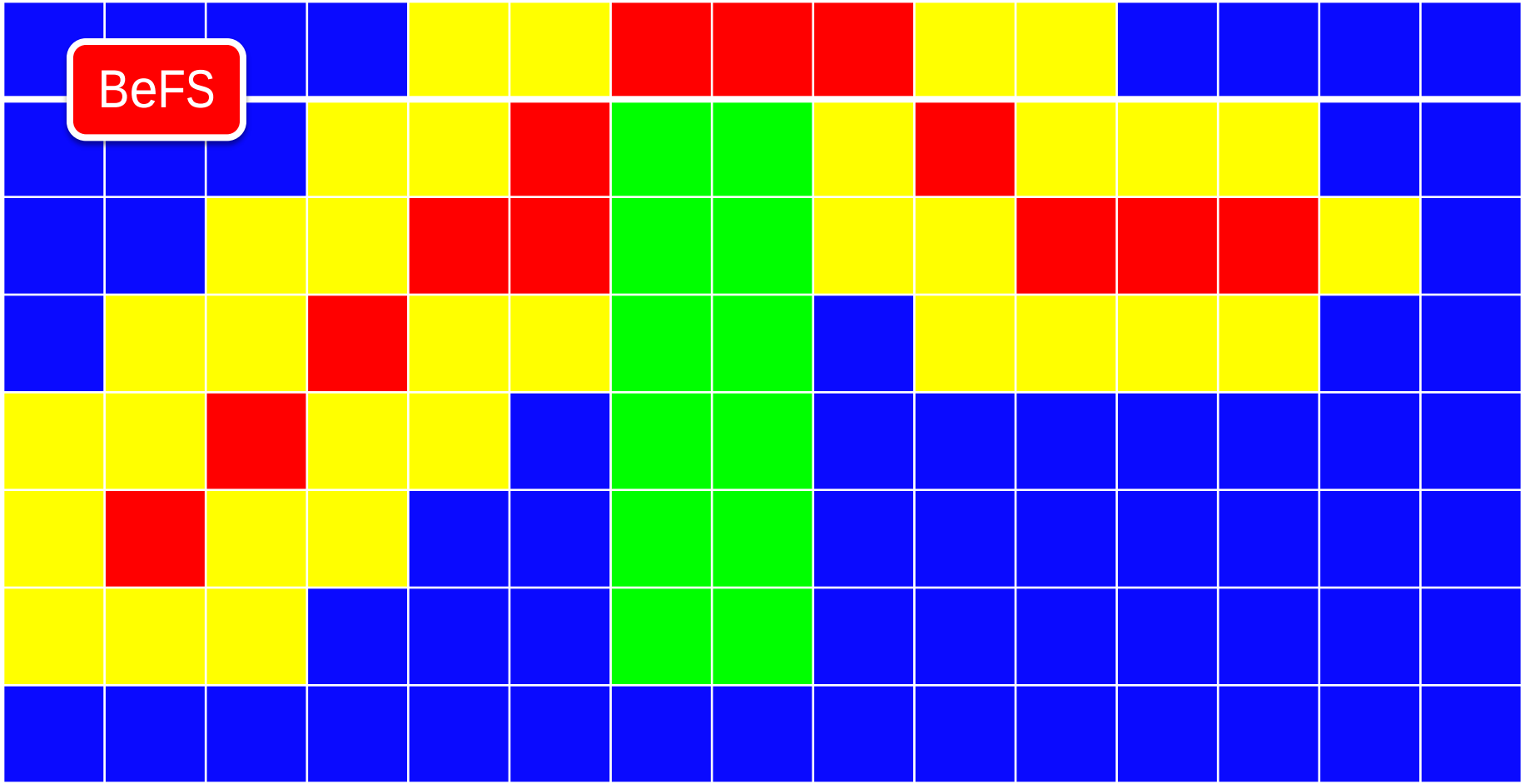

BFS

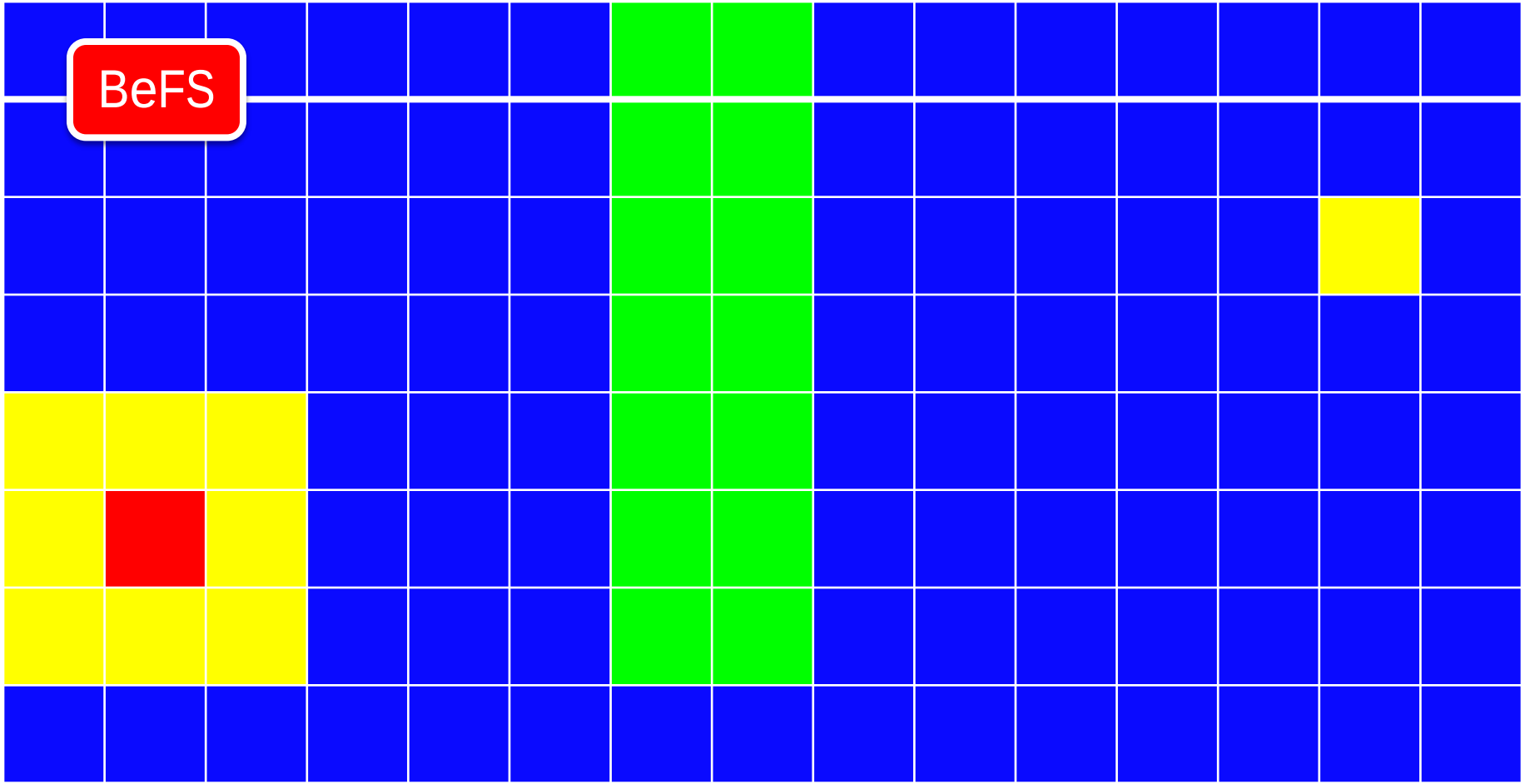

BeFS

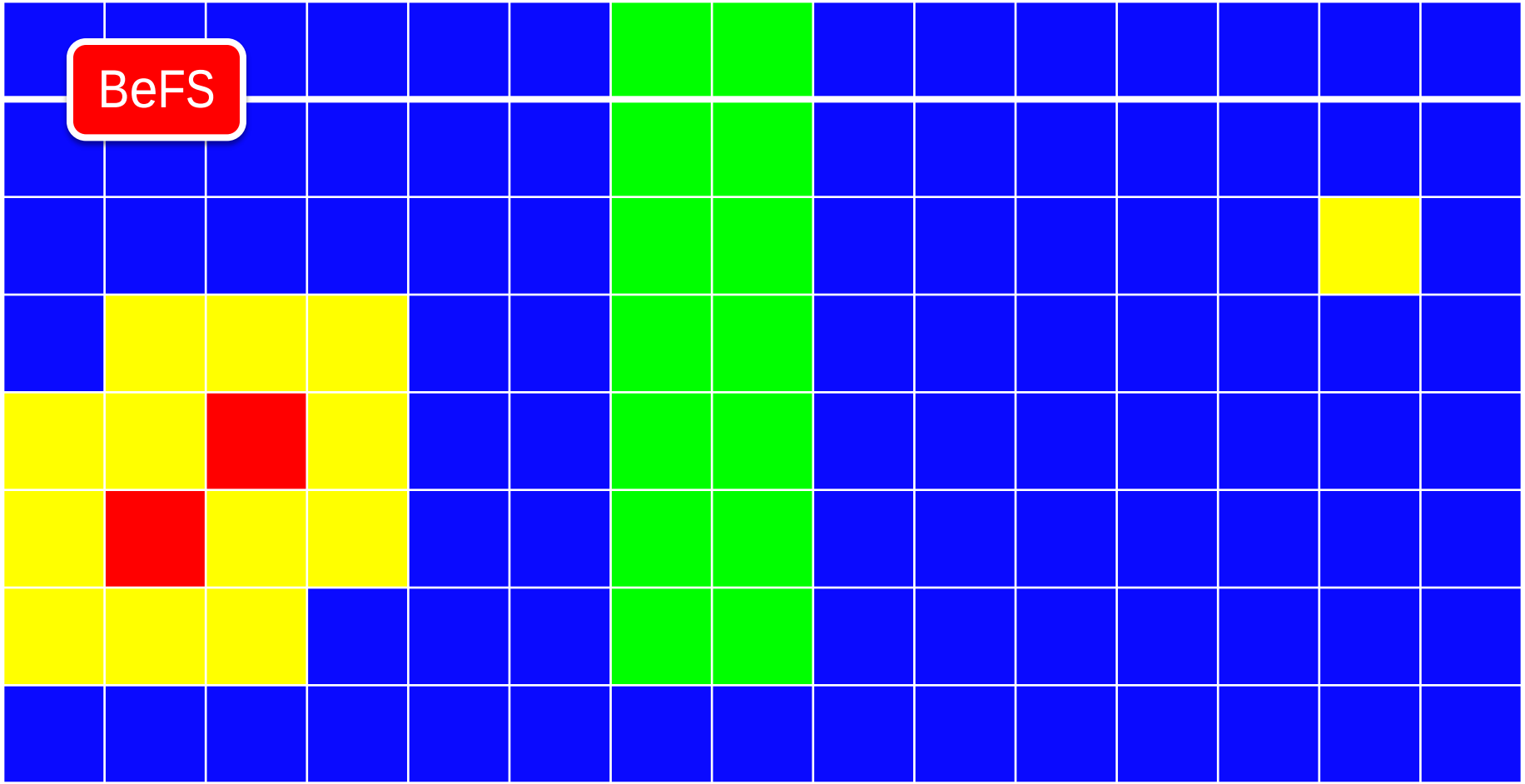

BeFS

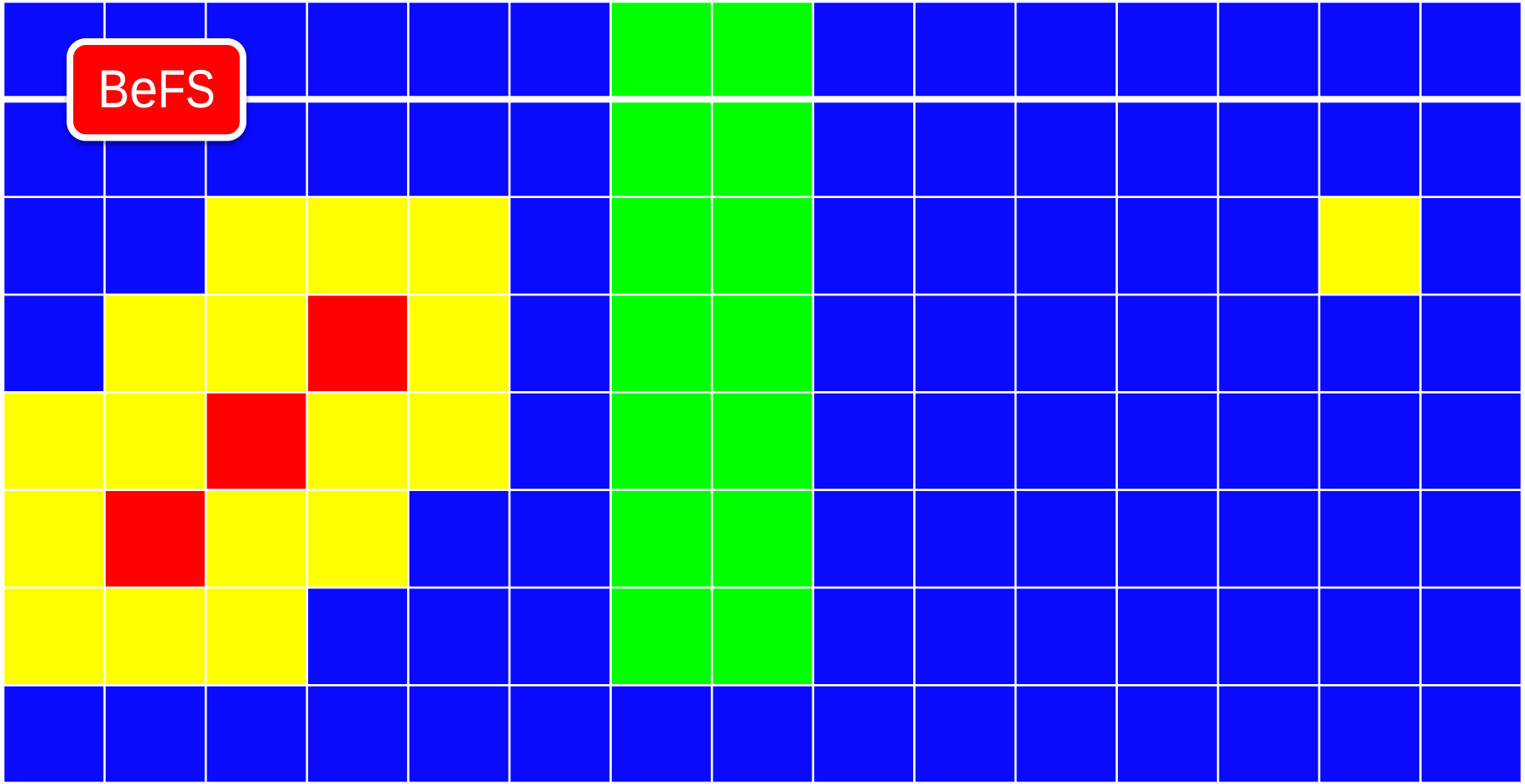

BeFS

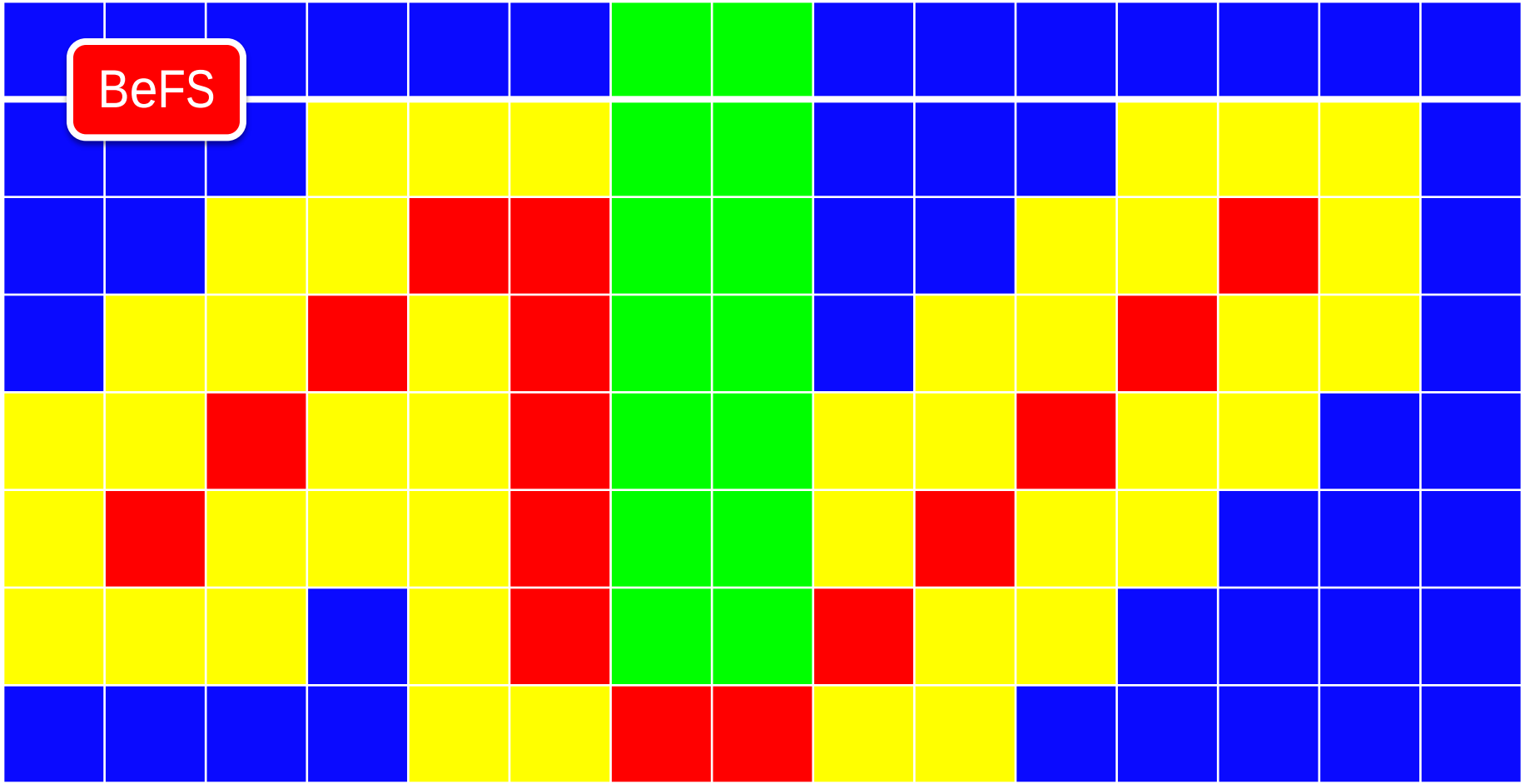

BeFS


BeFS


BeFS


BeFS


BeFS


BeFS

GBeFS Performance

- Suffers similarly to DFS
- Incomplete in General
  - May not find a goal
  - May get lost in paths if heuristic is bad
- Not Optimal
  - May find a higher cost path than optimal
- Time complexity $O(b^m)$, m = max depth

A* Search Algorithm

- Most well-known BeFS algorithm
- Evaluates nodes by using g(n) and h(n)
  - Heuristic function f(n) = g(n) + h(n)
  - f(n) = estimate of cheapest solution via n
- Select node from open list with min f(n)
- Performance depends on properties of the heuristic function

General Uninformed Tree Search

```
1. Function Tree-Search(problem, strategy)
2.     fringe = {Node(problem.initial_state)}
3.     while (true)
4.         if (fringe.empty) return fail
5.         node = strategy.select_node(fringe)
6.         if (node.state is goal) return solution
7.         else fringe.add(Expand(node, problem))
```

A* Tree-Search

```
1. Function AStar(problem, h(n))
2.     fringe = PriorityQueue(f(n)=g(n)+h(n))
3.     fringe.add(problem.initial_state)
4.     while (true)
5.         if (fringe.empty) return fail
6.         node = pop_min_f(fringe)
7.         if (node.state is goal) return solution
8.         else fringe.add(Expand(node, problem))
```

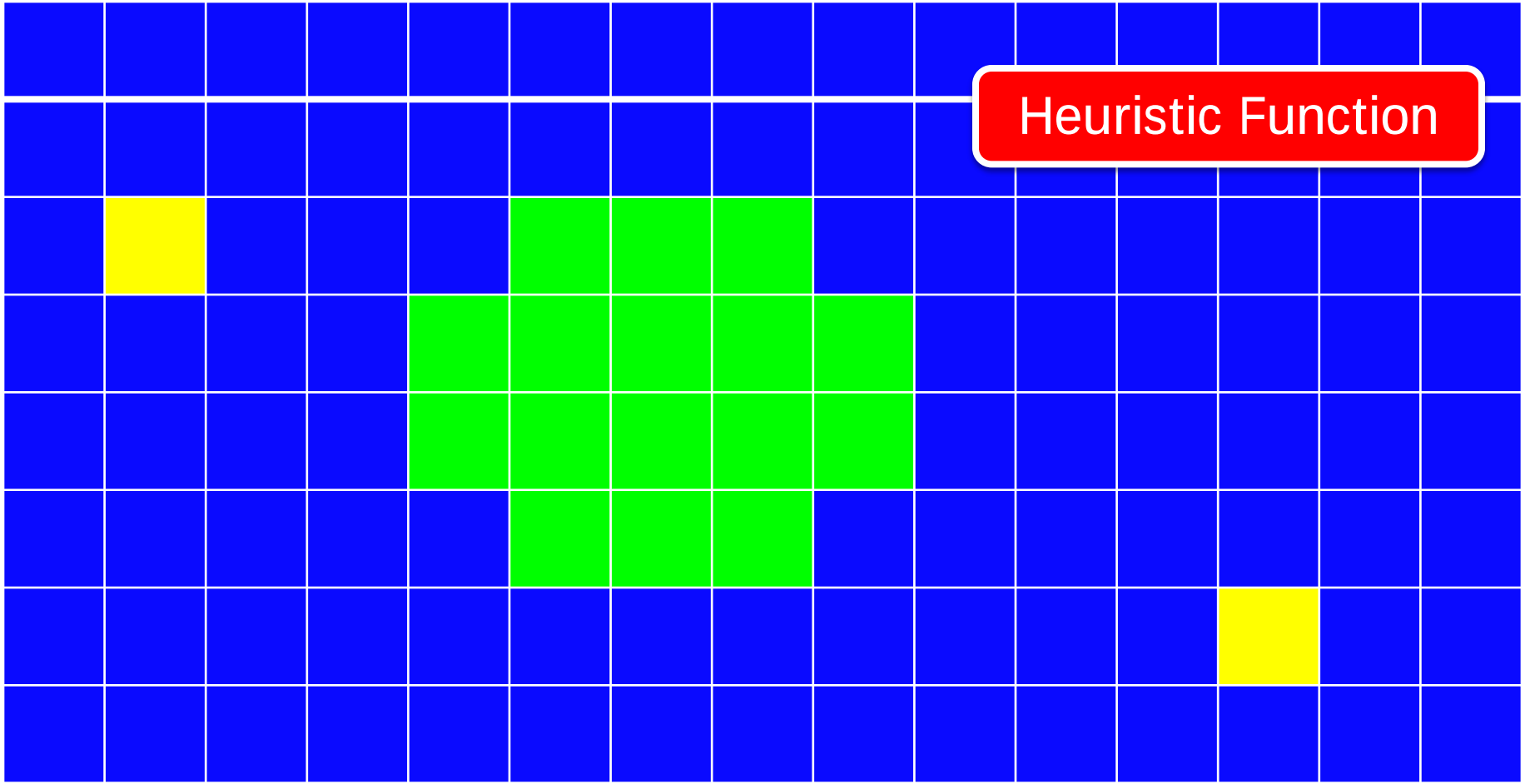
A* Performance

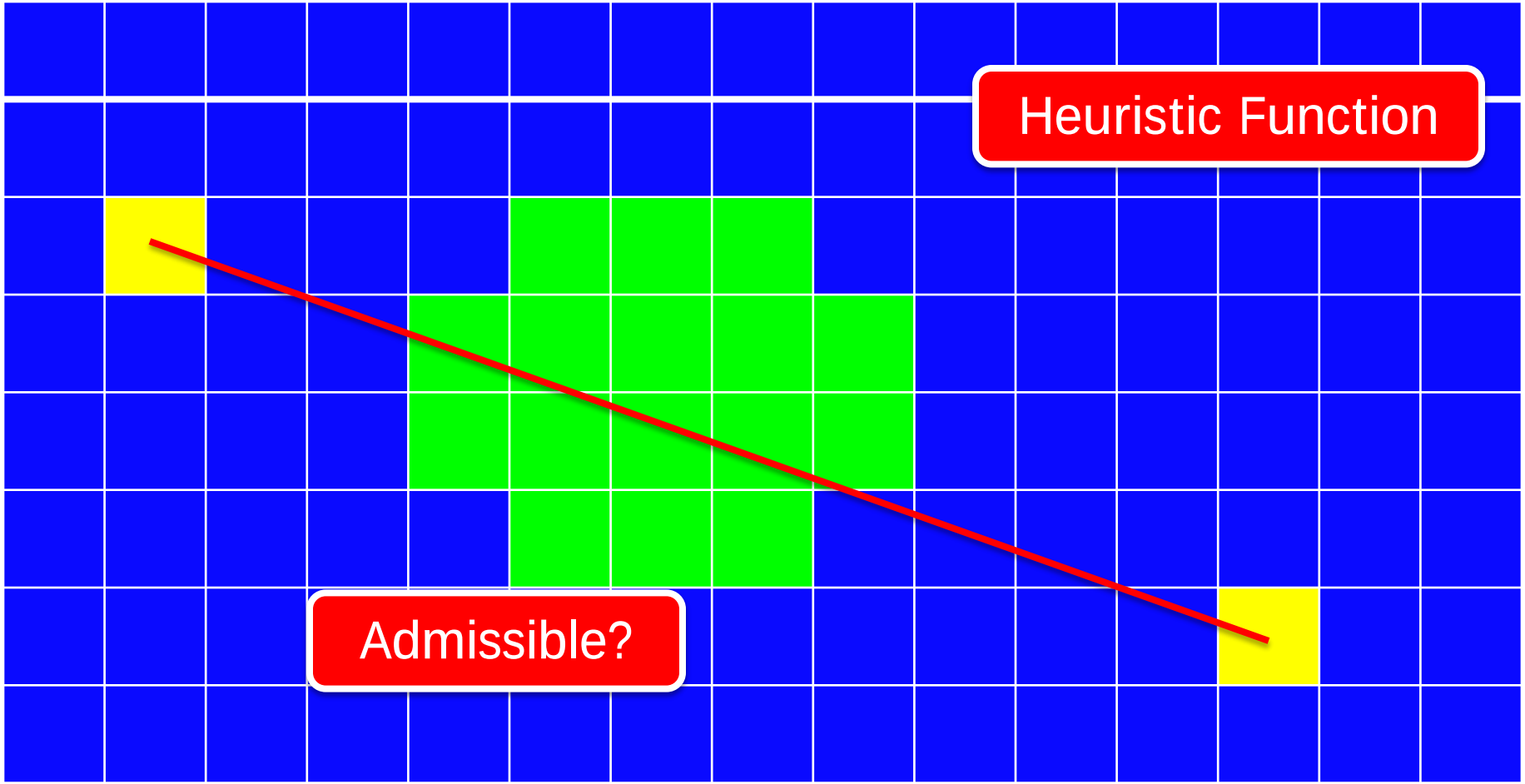
- A* using Tree-Search is complete
  - Will eventually search the entire tree, it will just search parts where the heuristic is lower first

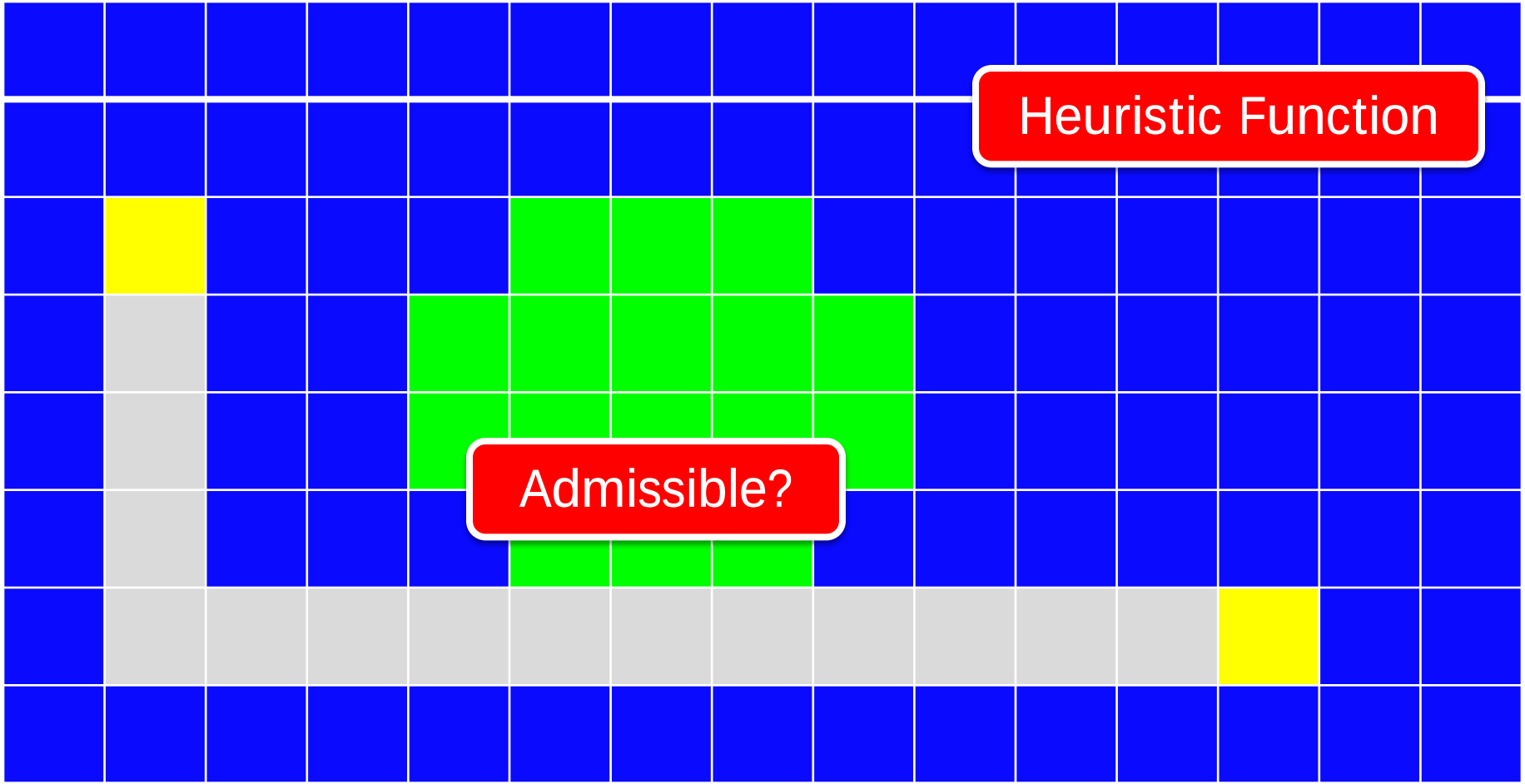
- A* using the Tree-Search algorithm is optimal if heuristic h(n) is admissible
  - Using a heuristic that is not admissible can (and usually does) produce non-optimal paths

Admissible Heuristic

- Never overestimates distance to goal
- Can be seen as ‘optimistic’ guesses
- h(n) <= h*(n)
- f(n) = g(n) + h(n) never overestimates true cost of a path through n when h(n) is admissible


Heuristic Function


Heuristic Function
Admissible?

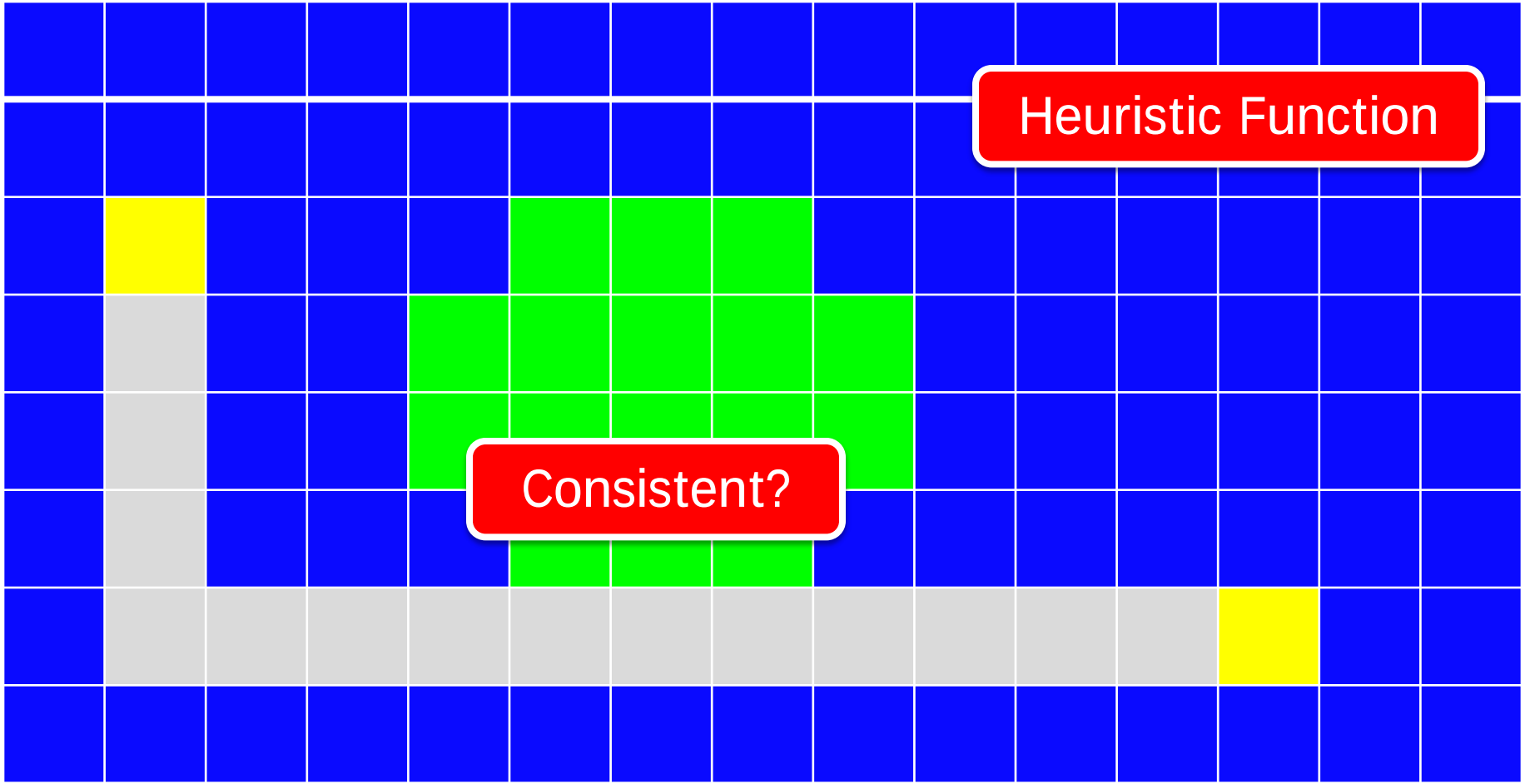

Heuristic Function
Admissible?

A* Using Graph-Search

- Can return suboptimal solutions
- A* Graph-Search using an admissible heuristic can discard an optimal path to a repeated state if it is not the first one that is generated during the search
- We can fix this by imposing a consistent heuristic

Consistent Heuristic

- Also called monotone heuristic
- Consistent if h(n) < c(n,a,n') + h(n')
  - h(n) = estimate path cost from n to goal
  - c(n,a,n') = cost of action a (transitions node n to n')
  - h(n') = estimate path cost from n' to goal
- "Estimate of reaching goal from n is no greater than the estimate of reaching the goal from n' plus the cost of getting to n' from n"


Heuristic Function
Consistent?

Consistent Heuristic

- Every consistent heuristic is also admissible
- Hard to construct an admissible heuristic that isn't consistent
- If h(n) is consistent, the values of f(n) along any path are non-decreasing
- The sequence of nodes expanded by A* using Graph-Search is in non-decreasing order of f(n)
- Therefore, first goal node selected for expansion is optimal, since all later nodes are at least as expensive
- A* using Graph-Search is optimal if h(n) is consistent

A* Graph-Search

```
1.  Function AStar(problem, h(n))
2.      closed = empty set
3.      open = PriorityQueue(Node(problem.initial_state), f=g+h)
4.      while (true)
5.          if (open.empty) return fail
6.          node = pop_min_f(open)
7.          if (node.state is goal) return solution
8.          closed.add(node.state)
9.          for child in Expand(node, problem)
10.             if (child.state in closed) continue
11.             child.f = child.g + h(n)
12.             open.add(child)
```

A* Graph-Search Performance

- Let C* be optimal solution path cost
- A* expands all nodes with f(n) < C*
- A* expands no nodes with f(n) > C*
- A* is optimally efficient for a given h(n)
  - No other optimal algorithm is guaranteed to expand fewer nodes than A*
  - Any algorithm not expanding all nodes with f(n) < C* risks missing the optimal path

A* Graph-Search Performance

- Pros
  - A* is Complete
  - A* is Optimal
  - A* is Optimally Efficient for given h(n)
- Cons
  - Number of nodes searched is still exponential in the length of the solution
  - For many large problems, A* is much better than uninformed search, but still infeasible in time / memory