# Laboratório de Programação Avançada 2018/19
# Week 3 – Backtracking

Universidade de Coimbra

# Outline

1. Introduction
2. Examples

Reading about problem solving with backtracking

- J. Erickson, Algorithms, Chapter 2
- J. Edmonds, How to think about algorithms, Chapter 17
- S.S. Skiena, M.G. Revilla, Programming Challenges, Chapter 8

## Backtracking

- Mostly used for optimization and constraint satisfaction problems.
- Backtracking uses recursion but stops it once an invalid partial solution is found, that is, extending this partial solution will always lead to an invalid/worse solution.
- Although it has the same time complexity of brute-force enumeration, it should be faster in practice
- We must carefully choose the **representation of the solution**.

## Backtracking template (for a decision problem)

---

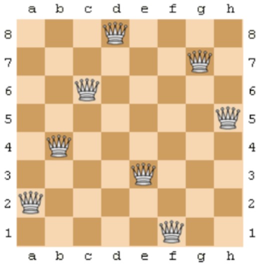
**Function** $BT(s)$

  **if** $reject(s) = \texttt{true}$ **then** $\{$rejection test$\}$
    **return** `false`
  **if** $accept(s) = \texttt{true}$ **then** $\{$base case$\}$
    $output(s)$
    **return** `true`
  **while** $condition(s) = \texttt{true}$ **do**
    $s' = update(s)$ $\{$new candidate solution$\}$
    **if** $BT(s') = \texttt{true}$ **then** $\{$recursive step$\}$
      **return** `true`
  **return** `false`

---

# Examples

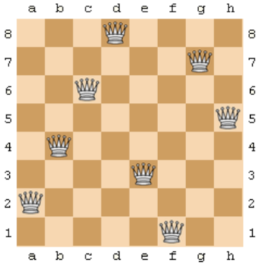
Problem: 8 Queens Problem




Find the position for 8 queens in a 8x8 chessboard such that no queen is able to capture another queen by using queen's move.

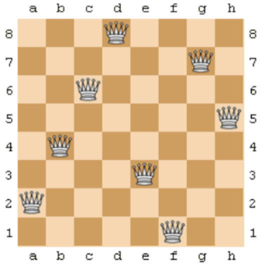
# Examples

Problem: 8 Queens Problem




Solution representation: Number all squares from 1 to 64.
The solution is a boolean list of 64 positions.

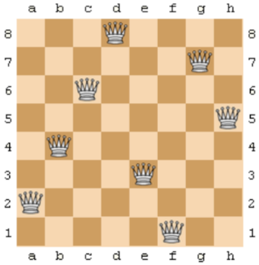
Problem: 8 Queens Problem




Solution representation: Number all squares from 1 to 64.
The solution is a boolean list of 64 positions.
This gives $2^{64} \approx 1.84 \times 10^{19}$ solutions!

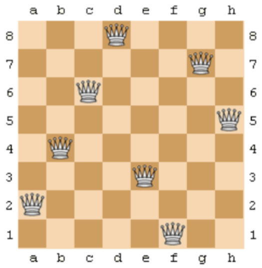
# Examples

Problem: 8 Queens Problem




Solution representation: List of 8 elements, one for each queen. Assign a cell number to queen *i* at position *i* in the list.

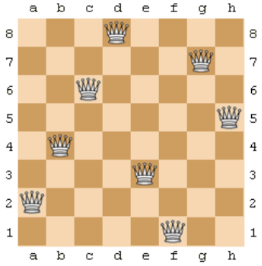
Problem: 8 Queens Problem




Solution representation: List of 8 elements, one for each queen.
Assign a cell number to queen $i$ at position $i$ in the list.
This gives $64^8 \approx 2.81 \times 10^{14}$ solutions!

# Examples

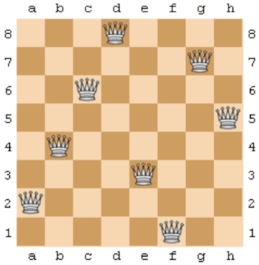
Problem: 8 Queens Problem




Solution representation: List of 8 elements, one for each queen. Two queens cannot be in the same column. Assign row number.

# Examples

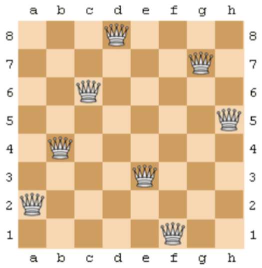
Problem: 8 Queens Problem




Solution representation: List of 8 elements, one for each queen. Two queens cannot be in the same column. Assign row number. This gives $8^8 \approx 1.67 \times 10^7$ solutions!

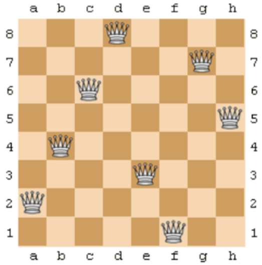
# Examples

Problem: 8 Queens Problem




Solution representation: List of 8 elements, one for each queen. Two queens cannot be in the same row and column: a permutation.

Problem: 8 Queens Problem




Solution representation: List of 8 elements, one for each queen. Two queens cannot be in the same row and column: a permutation.This gives $8! \approx 40320$ solutions!

# Examples

```
Function nQueens(col)
  if attack(col) = true then                    {rejection test}
    return false
  if col = N then                               {base case}
    return true
  for i = 1 to N do                             {for all unvisited rows}
    if row[i] = false then
      row[i] = true
      Q[col + 1] = i                            {assignment}
      if nQueens(col + 1) = true then           {recursive step}
        return true
      row[i] = false
  return false
```

## Examples

- Solution representation is given by list $Q$, which assigns a row number to a queen in a given column.

- At each recursive step, the queen $col + 1$ is tested in all empty rows.

- Function $attack(col)$ checks if queen at column $col$ is attacking any queen at column $j < col$.

- First call is $nQueens(0)$

# Examples

---

**Function** *nQueens*(*col*)

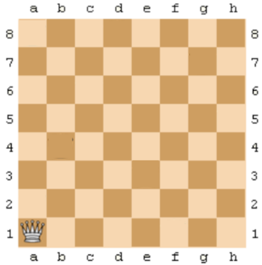
```
if col = N + 1 then                                   {base case}
    return true
for i = 1 to N do                                     {for all unvisited rows}
    if row[i] = false then
        row[i] = true
        Q[col] = i                                    {assignment}
        if attack(col) = false then
            if nQueens(col + 1) = true then           {recursive step}
                return true
        row[i] = false
return false
```

---

- Only feasible partial solutions are called recursively.
- First call is $nQueens(1)$

# Examples

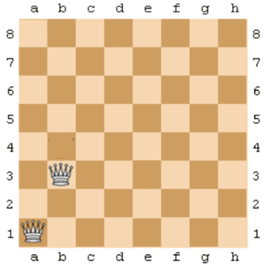
Problem: 8 Queens Problem




$$Q[a] = 1$$

# Examples

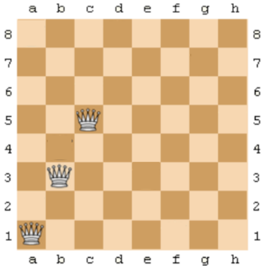
Problem: 8 Queens Problem




$$Q[b] = 3$$

# Examples

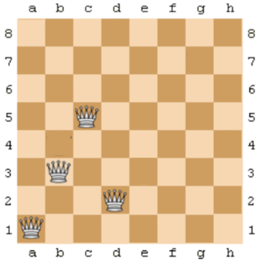
Problem: 8 Queens Problem




$$Q[c] = 5$$

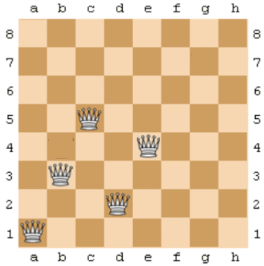
# Examples

Problem: 8 Queens Problem




$$Q[d] = 2$$

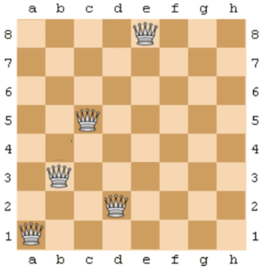
Problem: 8 Queens Problem




$Q[e] = 4$, but feasibility fails in column $f$.

# Examples

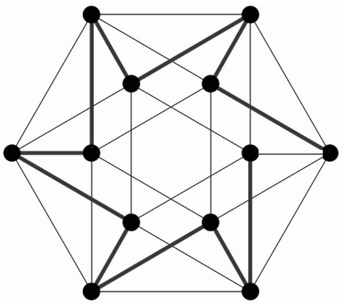
Problem: 8 Queens Problem




$Q[e] = 8$, but feasibility fails in column $f$. Backtrack!

# Examples

Problem: Hamiltonian path with starting and ending node



Given a graph $G = (V, E)$ and two nodes $s$ and $t$, determine whether a Hamiltonian path exists between node $s$ and $t$.

# Examples

```
Function HamPath(v)
  if v = t then                                   {base case}
    if sum(visit) = N then
      return true
    else                                          {rejection test}
      return false
  for each {v, i} ∈ E do                          {(implicit) rejection test}
    if visit[i] = 0 then
      visit[i] = 1                                {mark as visited}
      if HamPath(i) = true then                   {recursive step}
        return true
      visit[i] = 0                                {mark as unvisited}
  return false
```

- At each step, the insertion of edge $\{v, i\}$ is tested.
- Array `visit` marks the nodes that are visited.
- In the first call, mark node $s$ and call $HamPath(s)$.

# Examples

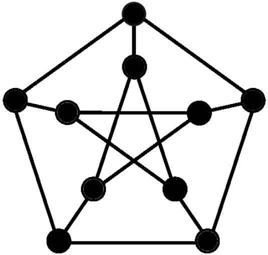
Problem: Hamiltonian cycle in graph $G = (V, E)$

```
Function HamCycle(v)
  if sum(visit) = N then                                  {base case}
    if (v, 1) ∈ G then
      return true
  for each {v, i} ∈ E do                     {(implicit) rejection test}
    if visit[i] = 0 then
      visit[i] = 1                                   {mark as visited}
      if HamCycle(i) = true then                      {recursive step}
        return true
      visit[i] = 0                                 {mark as unvisited}
  return false
```

- In the first call, mark node 1 and call $HamCycle(1)$.
- Check if it is a cycle at the base case.

# Examples

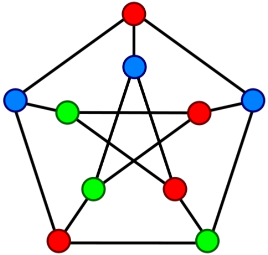
Problem: Graph coloring problem



Given a graph $G = (V, E)$ and $K$ colors, find whether you can color the nodes such that no two adjacent nodes have the same color.

# Examples

Problem: Graph coloring problem



Given a graph $G = (V, E)$ and $K$ colors, find whether you can color the nodes such that no two adjacent nodes have the same color.

# Examples

```
Function gcp(v)
  if v = N + 1 then                                   {base case}
    return true
  for i = 1 to K do                                   {for all colors}
    feasible = true
    for each {v, j} ∈ E do                            {rejection test}
      if color[j] = i then
        feasible = false
        break
    if feasible = true then
      color[v] = i                                    {assignment}
      if gcp(v + 1) = true then                       {recursive step}
        return true
      color[v] = 0                                    {remove assignment}
  return false
```

# Examples

- At each step, a feasible color is assigned to a node.
- $color[v] = 0$ means that node $v$ has no color yet.
- In the first call, color the first node and call $gcp(2)$.
- And if the goal is to minimize the number of colors?

# Examples

```
Function gcp(v)
  if v = N + 1 then                                   {base case}
    return 1
  c = 0
  for i = 1 to K do                                   {for all colors}
    feasible = true
    for each {v, j} ∈ E do                            {rejection test}
      if color[j] = i then
        feasible = false
        break
    if feasible = true then
      color[v] = i                                    {assignment}
      c = c + gcp(v + 1)
      color[v] = 0                                    {remove assignment}
  return c
```

Count how many colorings

Problem: The trip of Mr. Rowan

*Mr. Rowan plans to make a walking tour of Paris. However, since he is a little lazy, he wants to take the shortest path that goes through all the places he wants to visit. He plans to take a bus to the first place and another one back from the last place, so he is free to choose the starting and ending places. Can you help him?*

Problem: The trip of Mr. Rowan

*Mr. Rowan plans to make a walking tour of Paris. However, since he is a little lazy, he wants to take the shortest path that goes through all the places he wants to visit. He plans to take a bus to the first place and another one back from the last place, so he is free to choose the starting and ending places. Can you help him?*

It is the shortest Hamiltonian path problem (in a complete graph)

# Examples

**Function** $ShortPath(v, len)$

**if** $len \geq best$ **then** {1st rejection test}
  **return**
**if** $sum(visit) = N$ **and** $len < best$ **then** {base case}
  $best = len$
  **return**
**for each** $\{v, i\} \in E$ **do** {2nd rejection test}
  **if** $visit[i] = 0$ **then**
    $visit[i] = 1$ {mark as visited}
    $ShortPath(i, len + M[v][i])$ {recursive step}
    $visit[i] = 0$ {mark as unvisited}

- Array $M$ has the distance between every pair of locations.
- Variable *best* has the value of the shortest path found so far. It should be initialized with a large value.