COM3103

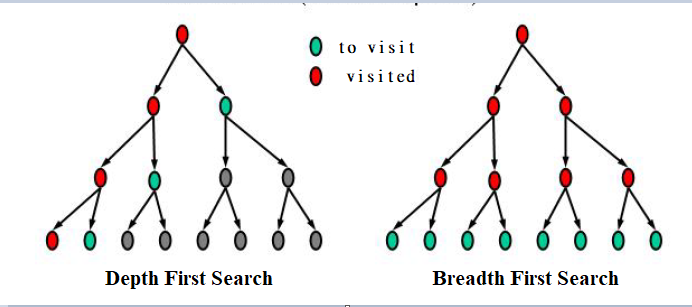
Artificial Intelligence

1. Advanced Search Algorithms:

Best First, A\* and CSP

**Recall from Chapter 4**

* Depth First Search
  + Search one branch completely to the end before other branches
* Breadth First Search
  + Search level by level



**Depth First Search vs Depth First Search**

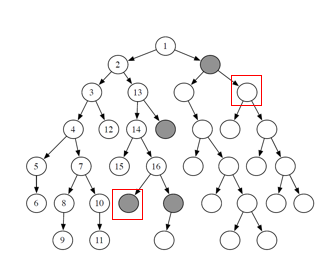
**Breadth first search**

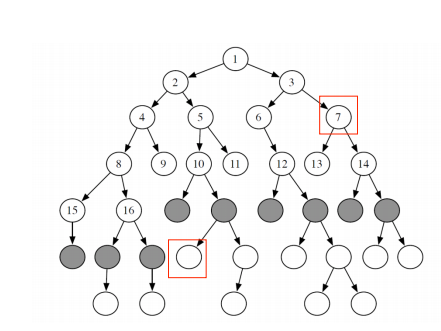
Advantage: Guarantee to find the shortest-path solution (the one nearest to the root)

Disadvantage: Exponential space requirement.

**Depth first search**

Advantage: Linear space requirement (if no need to check for visited nodes)

Disadvantage: May not find the shortest-path solution. (i.e., the solution may not be optimal)



**Which one to use, BFS or DFS?**

1. When we need the shortest path to a solution? BFS
2. When the solutions are hidden at great depth? DFS
3. The search graph is so big that some levels cannot fit into memory? DFS



**Comparison of DFS and BFS**

|  |  |  |
| --- | --- | --- |
|  | **Depth First Search** | **Breadth First Search** |
| Problems suitable | Solutions occur deep in the tree | Solutions occur at a reasonable level (not too deep) |
| Time  (worst case) | Potentially large (exponential)  *O(bm)* | Potentially large (exponential)  *O(bm)* |
| Memory (worst case) (for storing to-visit list) | Modest (linear). *O(b\* m)*  Need to store *to\_visit* nodes mainly in a single branch only at any time  (if no need to check for visited nodes) | Potentially large (exponential) *O(bm)*  Need to store all nodes of the current level in the to\_visit. |

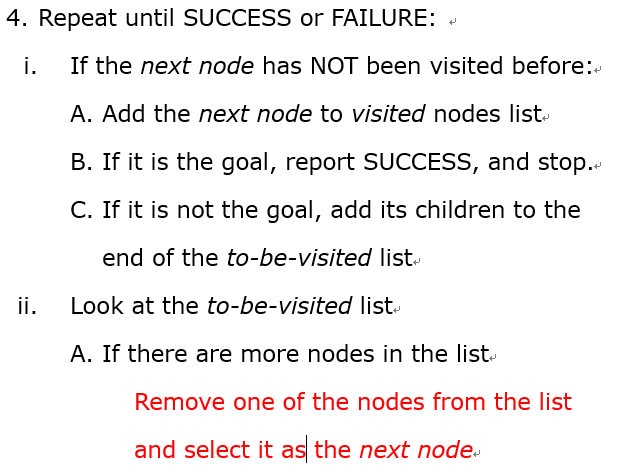
**Note**

1. *b* is the branch factor and *m* is the deepest level. *O(n)* means it is roughly proportional to *n*
2. All three algorithms will visit every node in the worst case. Time complexity could be high!

**Comparison of DFS and BFS**

|  |  |  |
| --- | --- | --- |
|  | **Depth First Search** | **Breadth First Search** |
| **Complete?** (Can always find solution) | No – if some branches are infinitely deep | Yes |
| **Optimal?** (shortest path always found?) | *No* | *Yes* |

**Recall: Basic Search Algorithm**



**Which node to remove from the to-visited list?**- DFS: last one in the list (LIFO)  
- BFS: first one in the list (FIFO)

**But, what if we have some ideas that some node may be better than the others?**

**Informed Search**

**Provide guidelines (heuristic) for selecting the next node to expand.**

**Heuristic** generally refers to a "rule of thumb", i.e., some "hints" that's helpful but not guaranteed to work.

Use some simple functions to estimate how good is each node in the to-visit-list (i.e., estimate how close it is to the goal).

There is no guarantee that the estimation is always correct.

Nevertheless, using heuristic functions may still speed up, at least on average, the process of finding a goal.

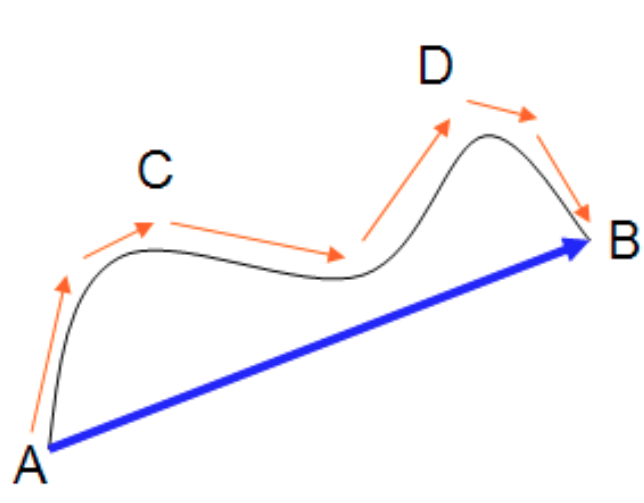
**Heuristic function**

A function that computes a value for a node, according to some heuristics

**Example: Estimated distance to goal**

A heuristic function that estimates the distance from a node to a goal.

E.g., using the straight-line distances between two cities to estimate the actual distance in a road network



**Best first search (greedy search)**

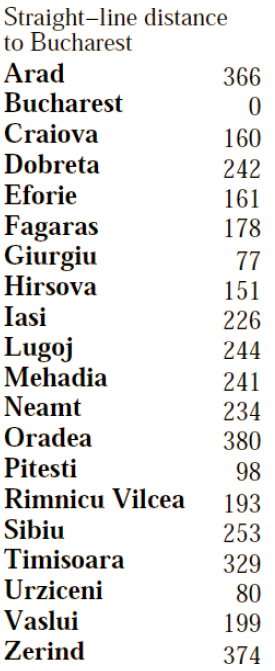
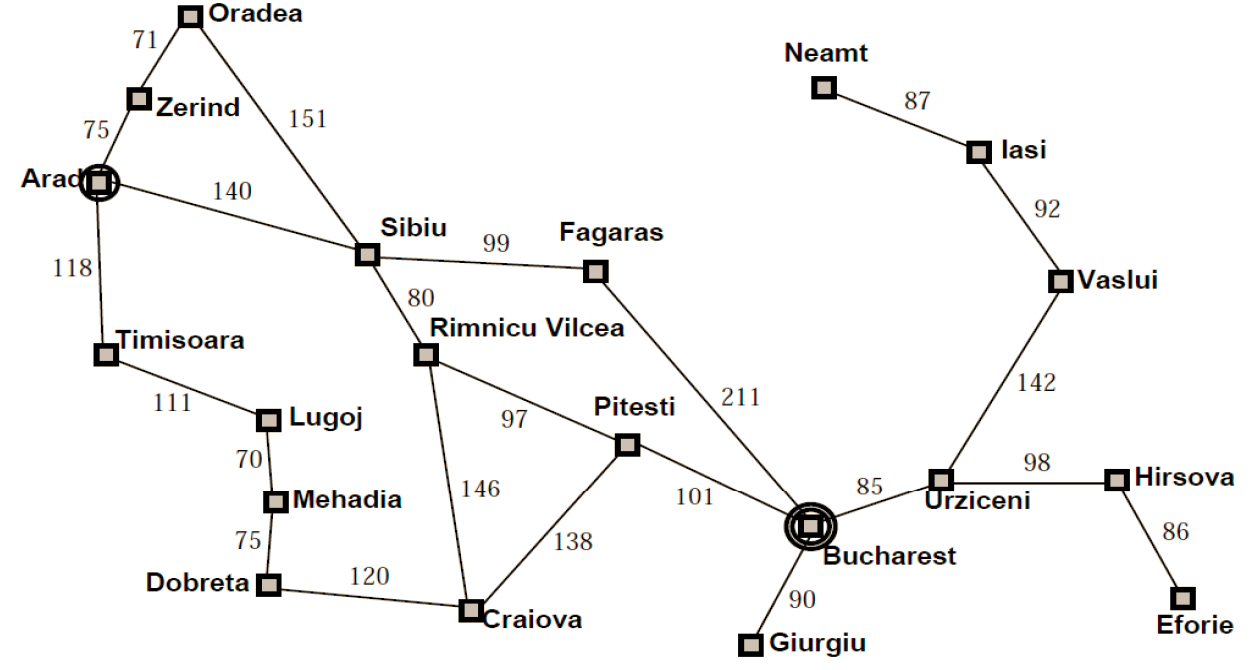
* + Evaluate every node *n* in the to-visit list using a heuristic function *h*(*n*)
  + Pick the one with the "best" *h*(*n*) scores as the next node one to visit
  + Note: for efficiency, we should keep track of the best scores in the *to-visit* list (how?)
    - Scan the whole list every time, while remembering the *h*(*n*) value of each node?
    - Keep the list sorted, and insert nodes in the right order?
    - other method (priority queue?)
  + Best-first has a kind of breadth-first flavor and we expect the *to-visit* list will tend to grow more than in depth-first search.
  + The worst-case performance of best first search is same as the breadth first search

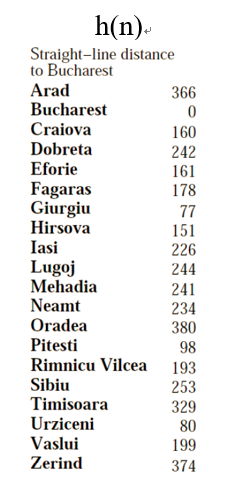
**Example of** **Best First Search: Romania Path Problem**

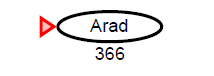
Find Shortest route from Arad to Bucharest

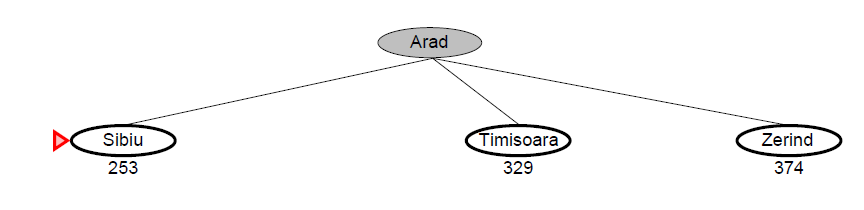
*h*(*n*) = straight-line distance from *n* to Bucharest

*h*(*n*)

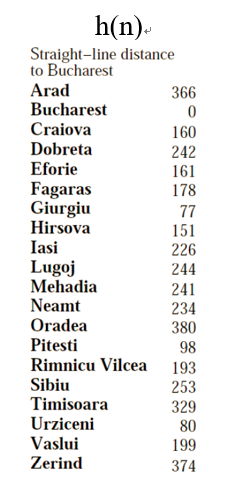
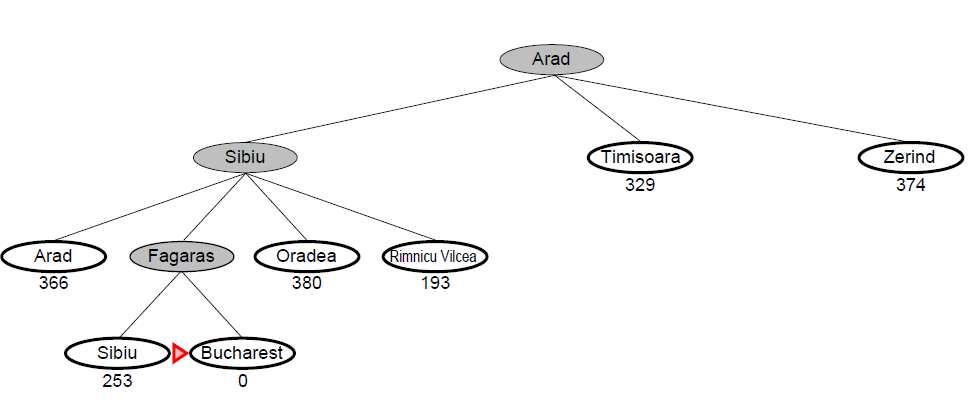
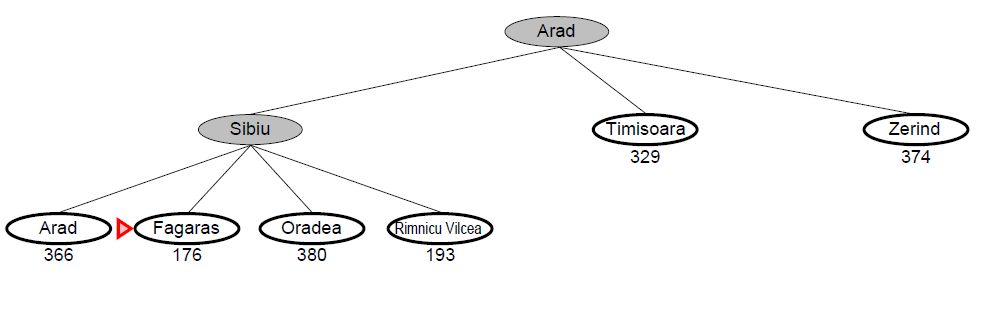


**Best First Search: Romania Path Problem**





**Best First Search: Romania Path Problem**



Best First Search (Greedy)Algorithm

1. Start at the root node (called the *next node*).
2. Setup a list of *visited* nodes (initially empty)
3. Setup a list of *to-visit* nodes (initially empty)
4. Repeat until SUCCESS or FAILURE:
   1. If the *next node* has NOT been visited before:
5. Add the *next node* to *visited* nodes list
6. If it is the goal, report SUCCESS, and stop.
7. If it is not the goal, add its children to the *to-visit* list together with the heuristic scores of each child-state
8. Look at the *to-visit* list
9. If there are more nodes in the list

Remove the node *n* with the lowest heuristic score from the list and select it as the *next node*

1. Else if no more nodes in the list

Report FAILURE

|  |  |  |  |
| --- | --- | --- | --- |
|  | **DFS** | **BFS** | **Best First Search** |
| **Complete?** (Can always find solution) | No – if some branches are infinitely deep | Yes | Yes |
| **Time (worst case)** | *bm* | *bm* | *bm* |
| **Space (worst case)** | *bm –* if no need to check for visited nodes | *bm* | *bm* |
| **Optimal?** (shortest path) | *No* | *Yes* | *Yes* |

**Comparison of uninformed search algorithms**

* On the average case, however, Best First Search can speed up the search provided a good heuristic is provided.
* However, it should also be noted that Best Search needs more effort to select the next-node comparing with BFS and DFS

**A\* Search**

* Based on best-first search
* Also consider the (actual) cost from root to the next stop in addition to the heuristic function (unlike the heuristics, the root to next stop cost is accurate)
* Evaluation function: *f*(*n*) = *g*(*n*) + *h*(*n*)

*n*: a candidate next node

*g*(*n*): the cost function (actual cost from root to the next node *n*)

*h*(*n*): the heuristic function (estimated cost from *n* to the goal (the heuristics))

Note: A\* search becomes best first search if g(n) = 0 for all n.

**Example of A\* Search: Romania Path Problem**

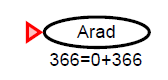
Evaluation function: *f*(*n*) = *g*(*n*) + *h*(*n*)

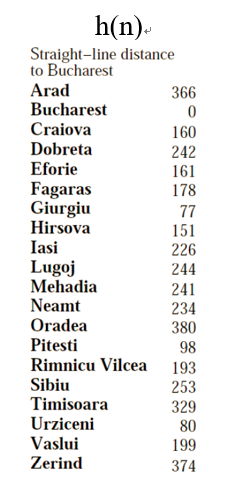
*n*: a city

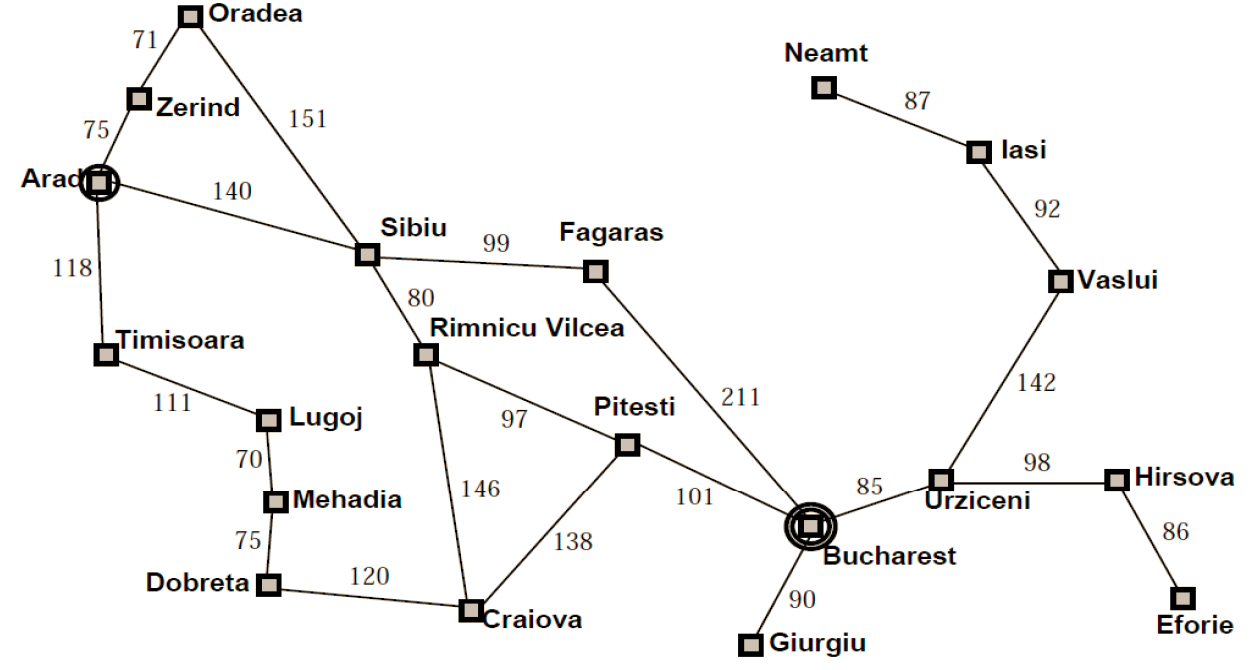
*g*(*n*): the actual road distance from the starting location (Arad) to *n*

*h*(*n*): *estimated* distance from *n* to target (the straight line distance)

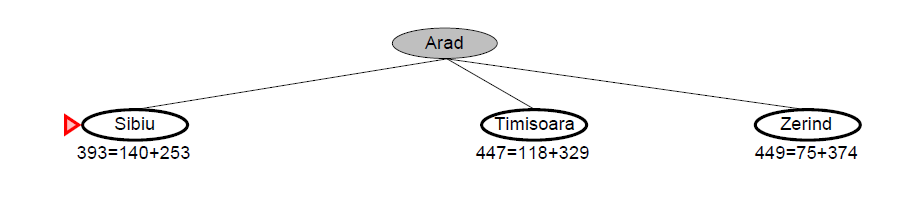
**Example of A\* Search: Romania Path Problem**

We start from Arad.

h(n) = 366

g(n) = 0.   
f(n) = g(n) + h(n) = 360

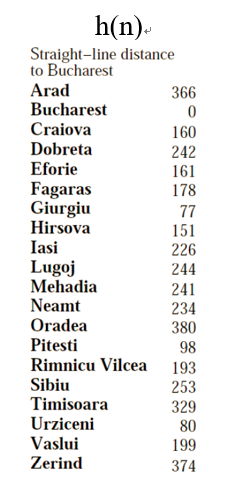
**Example of A\* Search: Romania Path Problem**

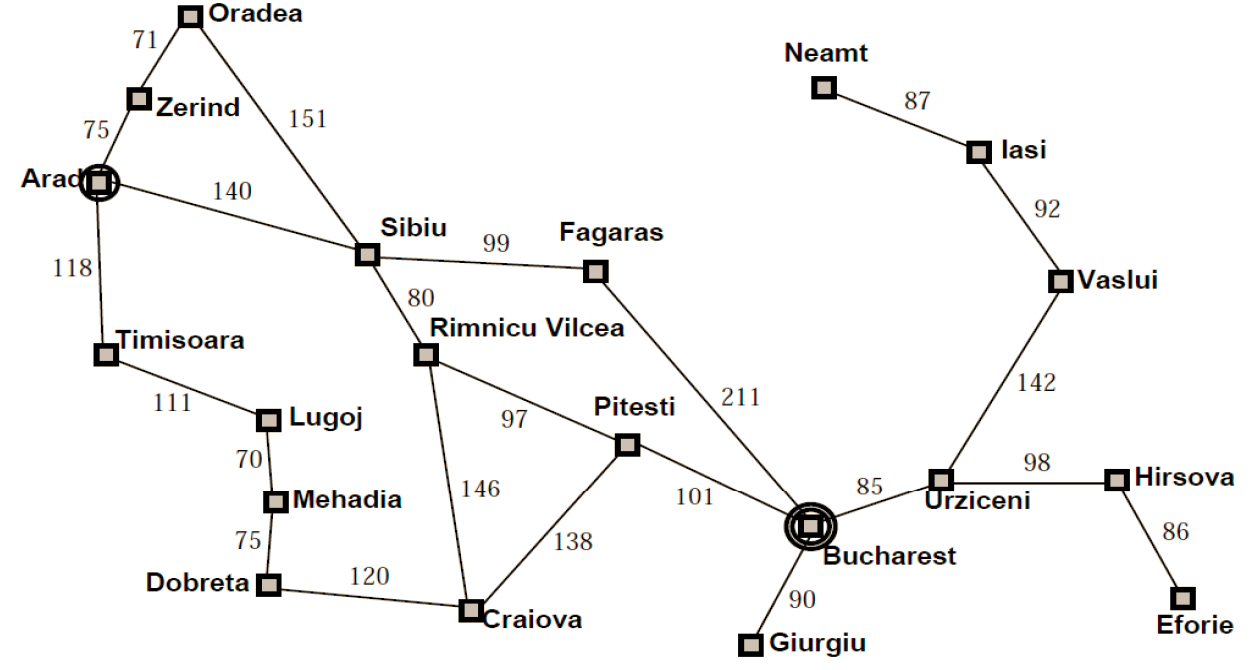


75

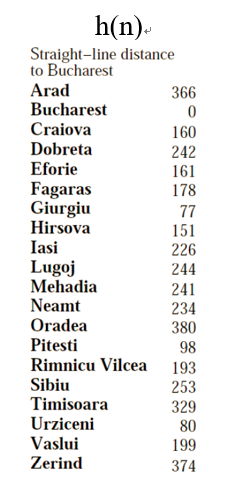
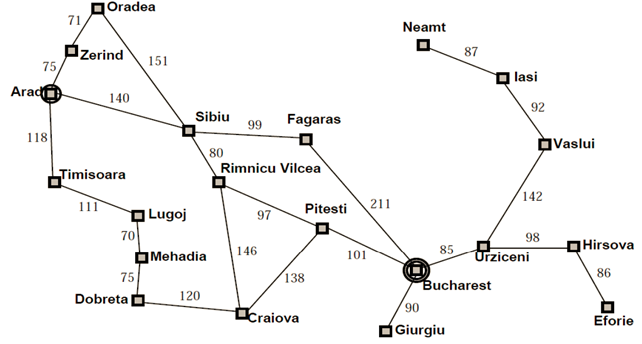
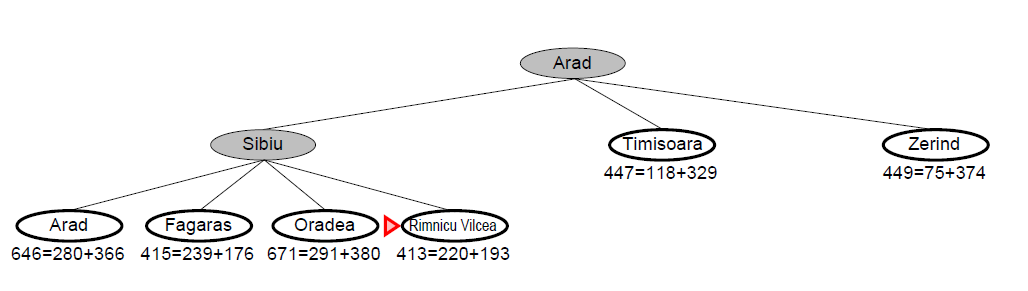
140

118

Sibiu: g(n) = 140, h(n) = 253  
Timisoara: g(n) = 118, h(n) = 329  
Zerind: g(n) = 75, h(n) = 374



**Example of A\* Search: Romania Path Problem**



118

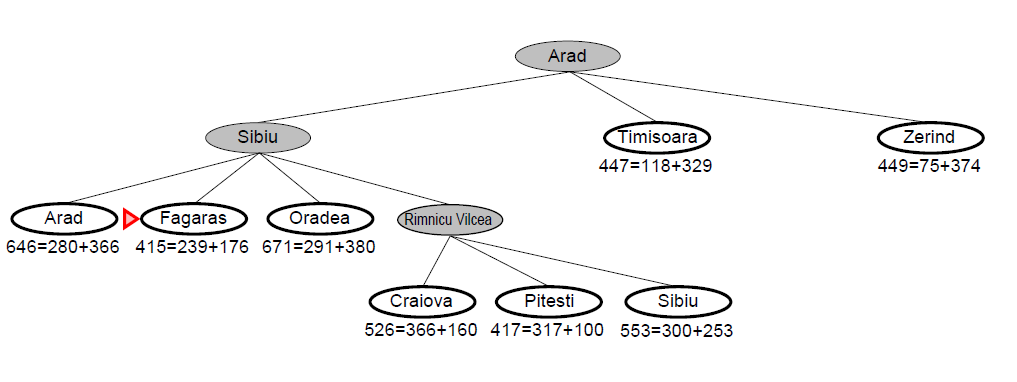
99

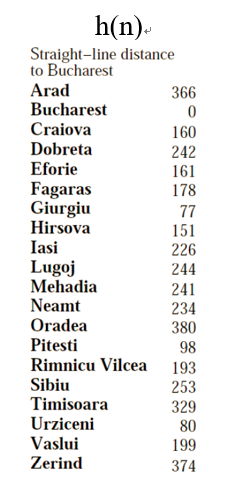
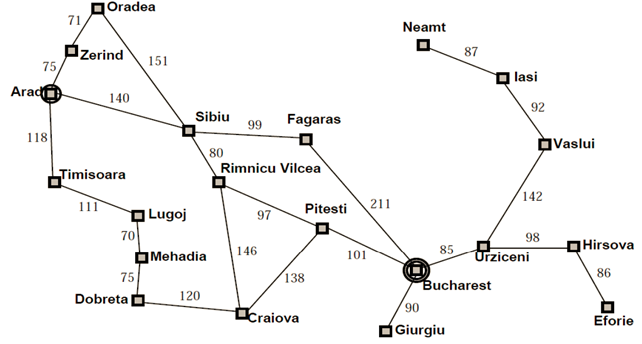
80

140

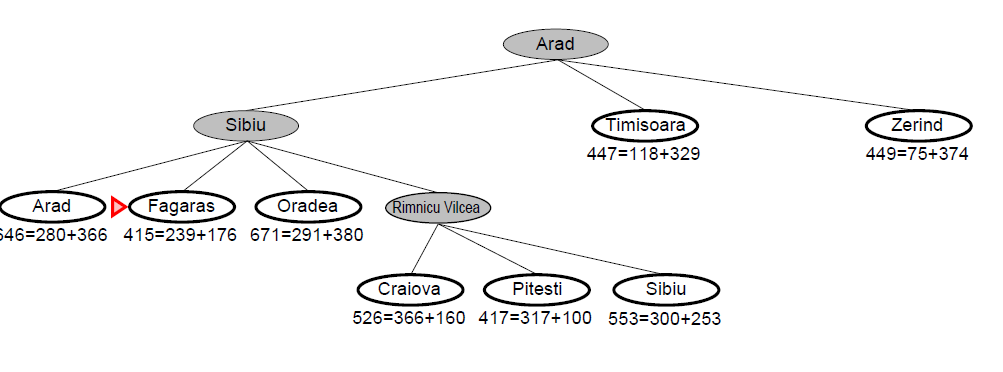
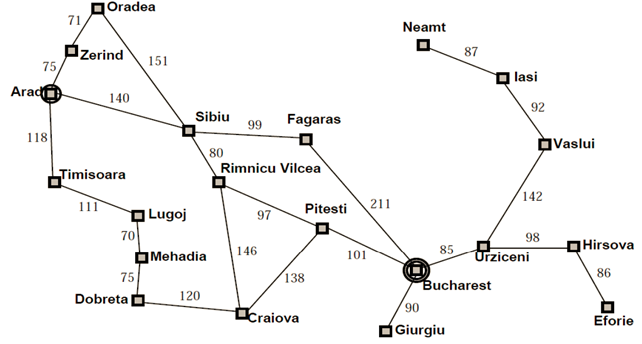
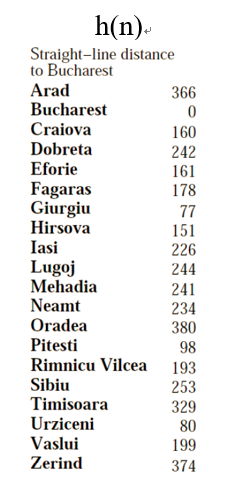
75

140

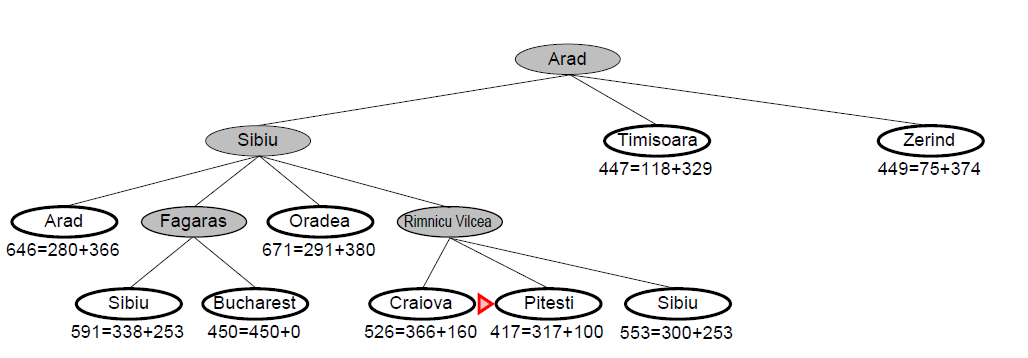
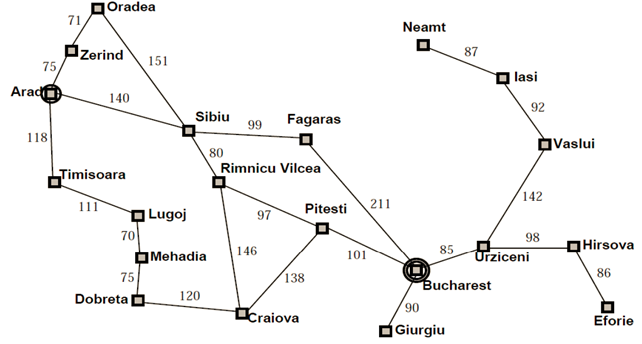
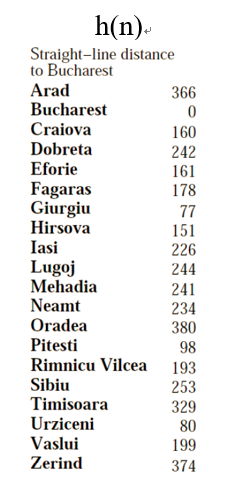
**Example of A\* Search: Romania Path Problem**



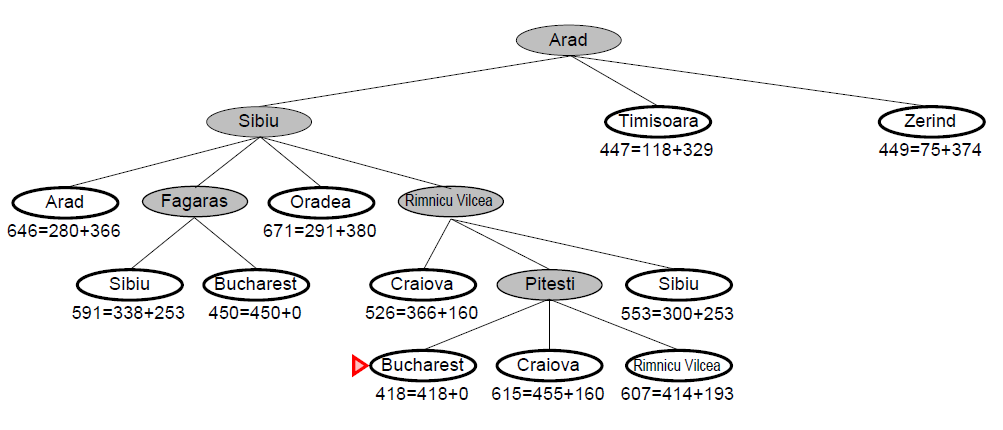
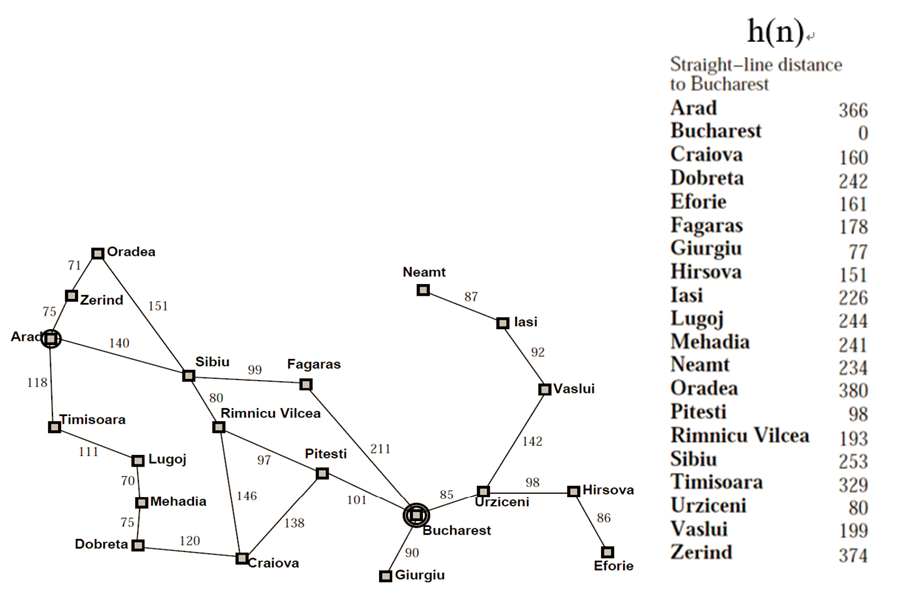
**Example of A\* Search: Romania Path Problem**



**Example of A\* Search: Romania Path Problem**



**Example of A\* Search: Romania Path Problem**



A\* Algorithm

**Given function:**

**cost(n1, n2)** – the cost of getting from *n*1 to *n*2  
**h(n)** – the estimated cost getting from *n* to the goal

**Steps:**

1. Let the *starting state* be the *next state*.
2. Setup a list of *visited states* (initially empty)
3. Setup a list a *to-visit* nodes (initially empty),
4. Let g(next state) = 0
5. Repeat until SUCCESS or FAILURE:
   1. If the *next state* has NOT been visited before:
6. Add the *next state* to *visited* *states* list. The *next state* is now the *current state*.
7. If it is the goal, report SUCCESS, and stop.
8. If it is not the goal
   * + - for each *child* of the *current state*, do

g(*child*) = g(*current\_state*) +

*cost*(*current\_node, child*)

Add the *child,* together with the g(*child*)and h(*child*), to the *to-visit* list

1. Look at the *to-visit* list
2. If there are more nodes in the list

Remove from the list the node *n* with the lowest evaluation score f(n) = g(n) + h(n)

and select it as the *next node*

1. Else If no more nodes in the list

Report FAILURE

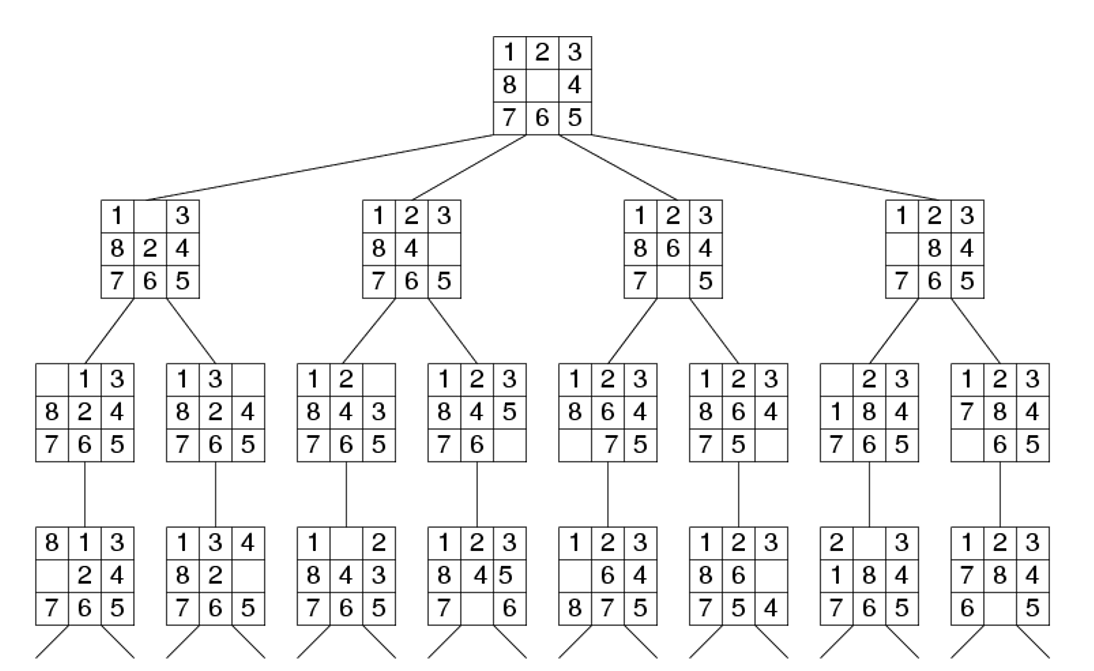
**Is A\* optimal? (can find the best solution)?**

It is optimal if the heuristic function *h*(*n*) never overestimates the cost to reach the goal (or we say, *h*(*n*) is *admissible*)

**Admissible**: the estimated cost *h*(*n*) is never higher than the real cost.

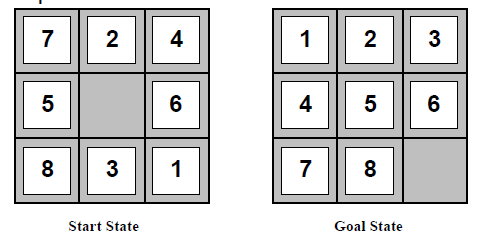
In the Romania path problem, the straight-line distance is always lower than the actual road distance. Therefore, it is admissible.

**Example: 8-puzzle**



Question: If the average branch factor is 3, and a solution is located at level 15 of the tree, how many nodes need to be searched in the worst case using BFS and DFS? 315

**Solving 8-puzzles problem using A\***



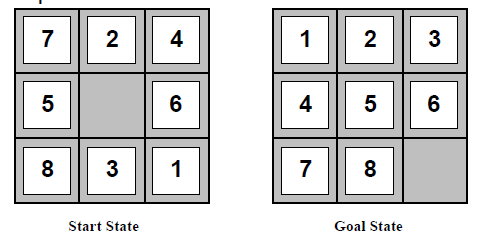
State: a list showing the position of the pieces. e.g. [7,2,4,5,0,6,8,3,1]

Goal state: the target position of the pieces, e.g. [1,2,3,4,5,6,7,8,9,0]

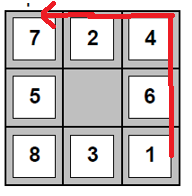
f(n) = g(n) + h (n)

Cost function g(n): 1 per move. (Each move costs the same)

h(n): ??

**Solving 8-puzzles problem using A\***

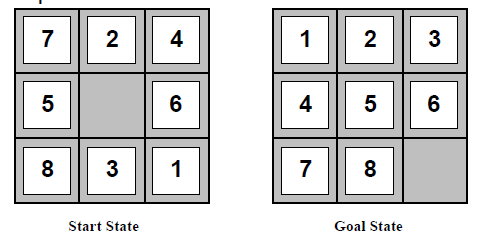
Consider two heuristics, h1(n) and h2(n)

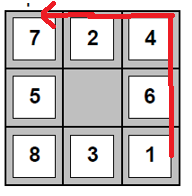
Let h1(n) = number of files in wrong position

Let h2(n) = total Manhattan Distance (the sum of horizontal and vertical distances).

h1(n) = 6

h2(n) = 4+0+3+3+1+0+2+1=14

 **Which heuristic is better? (Dominance)**

Let h1(n) = number of files in wrong position

Let h2(n) = total Manhattan Distance (the sum of horizontal and vertical distances).

* Both h1 and h2 are admissible.
* In general, given two admissible heuristic h1 and h1. If h2(n) >= h1(n) for all n, then we say h2 *dominates* h1, and h2 is the better heuristic and requires fewer iterations (and hence faster).  
  Which heuristic is better for the 8-square problems?

h1 (number of wrong tiles)?

h2 (Manhattan distance)?

**Exercise 1**

Writing a program for solving the 8-puzzle problem.

Test your program using the following problem instances using both *number of wrong tile heuristic* (h1) and *Manhattan distanc*e heuristic (h2)

[1,2,3,4,0,5,6,7,8] # problem 1: easy

[5,1,3,4,6,8,7,0,2] # problem 2: easy

[6,4,3,1,8,0,5,2,7] # problem 3: medium

[1,7,4,2,3,8,6,0,5] # problem 4: hard

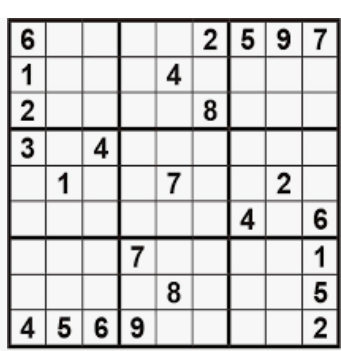
[4,7,2,0,8,1,3,6,5] # problem 5: hard

For each case, report the running time for both heuristics as well as whether the solution has been found.

**Constraint Satisfaction Problems (CSP)**

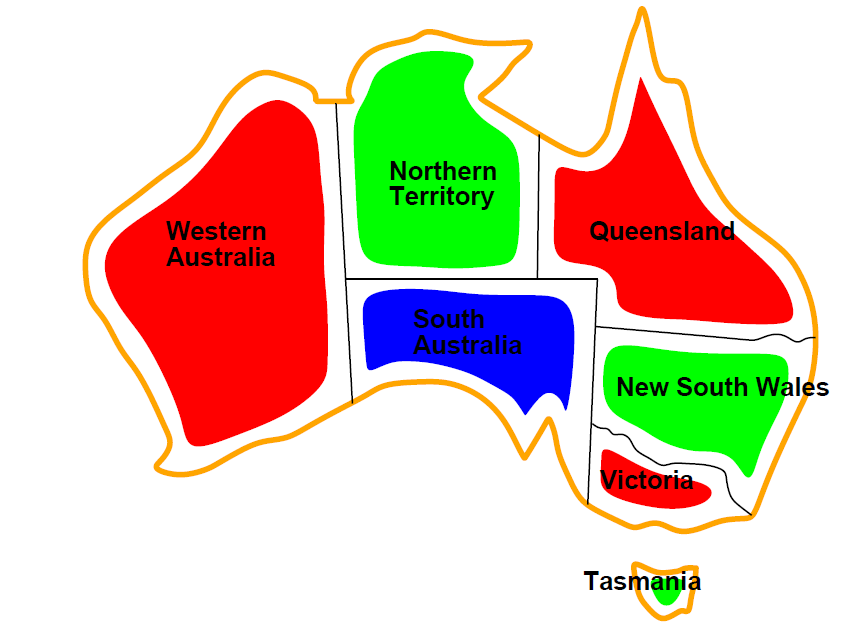
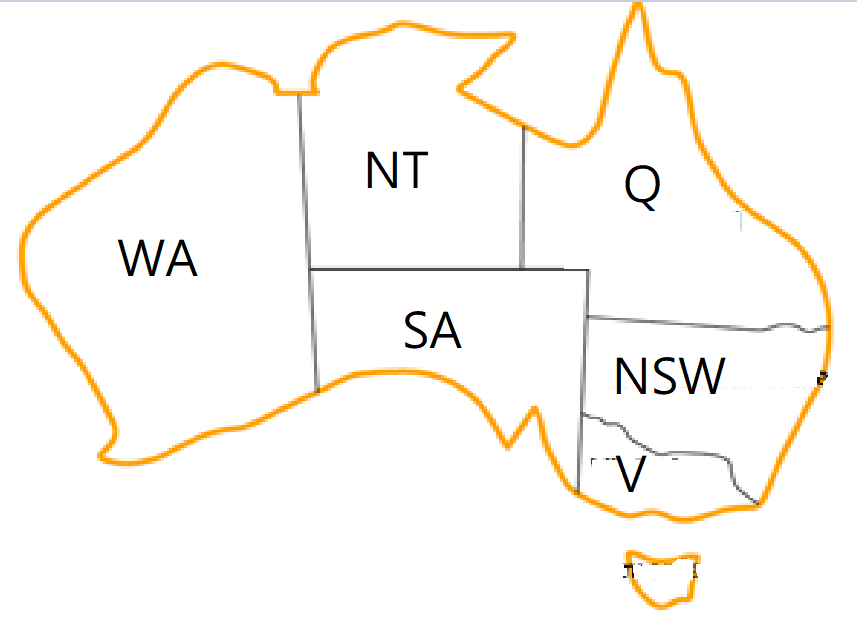
* Given:
  + Some **Variables** [*x*1, *x*2, …, *x*n]
  + Each variable *x*i may has some allowed values (called the **Domains**)
  + There are some **Constraints** that the values must satisfy
* Question: what should be the values of each variable, so that the constraints are not violated?

**Example applications**

* Time table assignment (classes must to assign to room and time without conflicts)
* Floorplanning (pieces need to be placed on limited space without overlapping)
* Sudoku

**Example: Map Colouring**

Problem: assign colour to various part of a map so that no two adjacent parts have the same colour.



**Example**:

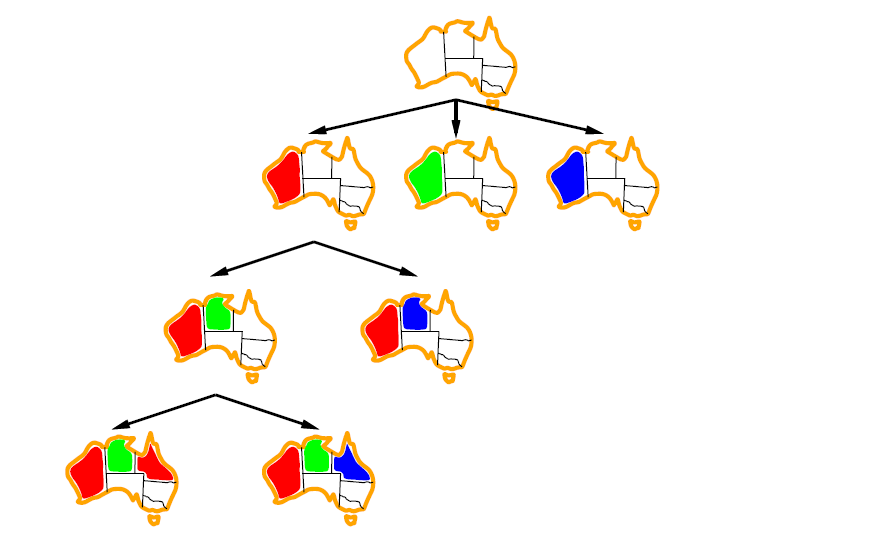
Australia

* Variables: [WA, NT, SA, Q, NSW, V, T]
* Domain = {red, green, blue}
* Constraints: no two neighbouring regions have the same colour, i.e,

WA ≠ NT, WA ≠ SA, NT ≠ Q, NT ≠ SA, SA≠Q, Q ≠ NSV, SA ≠ NSW, NSW ≠ V, SA≠V

**Searching for CSP solution**

* Tree-based representation
* Assign one variable to possible values at each level of the tree
* If constraint violated, do not consider the remaining of that branch anymore ("Back tracking").
* Solution occurs at the last level of the tree
* **Question:** Which tree searching algorithm is suitable? DFS

 🡸 starting state

🡸 Assign values for WA

🡸 Assign values for NT

🡸 Assign values for Q

**Searching for CSP solution**

WA NT SA

[None, None, None, ….]

WA NT SA

WA NT SA

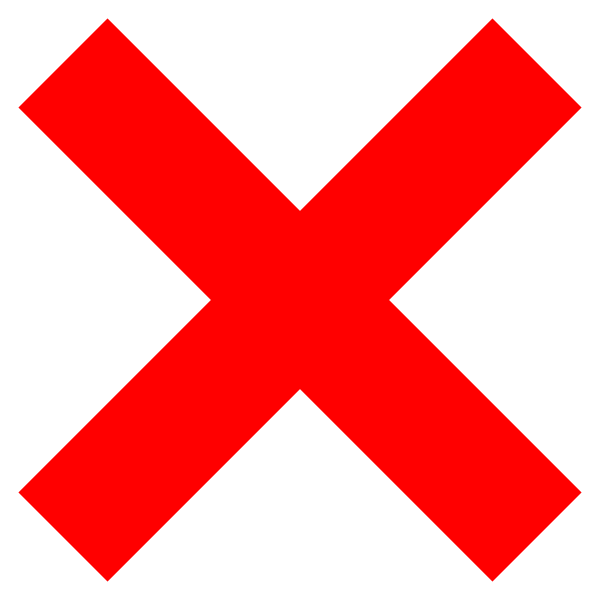
WA NT SA

[Red, None, None, ….] [Green, None, None, ….] [Blue, None, None, …]

WA NT SA

WA NT SA

WA NT SA

[ Red, Red, None, ….] [Red, Green, None, ….][Red, Blue, None, …]

**Constraint Violated!**

CSP Algorithm

**Need to define:**

1. A list of Variables
2. A list of allowed values for the variables (the Domain)
3. Functions for checking the Constraints

**Steps:**

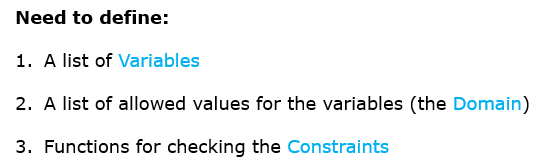
1. Assign NONE to each variable
2. Setup a list a *to-visit* states (initially empty),
3. *next\_state* = a list of current values of each variable (all have the value NONE initially)
4. Repeat until SUCCESS or FAILURE:
5. The *next\_state* is now the *current\_state*
6. If the *current\_state* is a solution, report SUCCESS, and stop.
7. If it is not a solution
   * + 1. Create a list of *children* by assigning each possible value to the next variable that still has NONE value
       2. For each child in the list of *children*:

If a *child* has **no constraint violation** with the previously assigned variables in the *current\_state*, append *child* to the to-visit list

1. Look at the *to-visited* list
2. If there are more nodes in the list
   * + 1. Remove the last node *n* and select it as the *next node*
3. Else If no more nodes in the listf
   * + 1. Report FAILURE

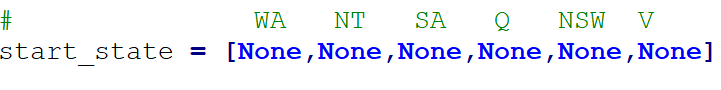
**Example: Map Colouring (Australia)**

**Three things we need to define for each problem:**

****

1. Defining a list of Variables

* The variables are just represented by the positions in the state.



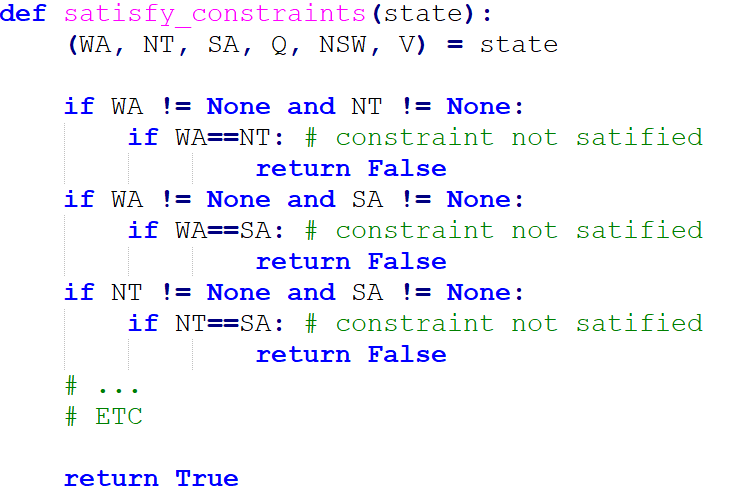
* Note that each element in a state is the assigned value of a variable (or None of not yet assigned)

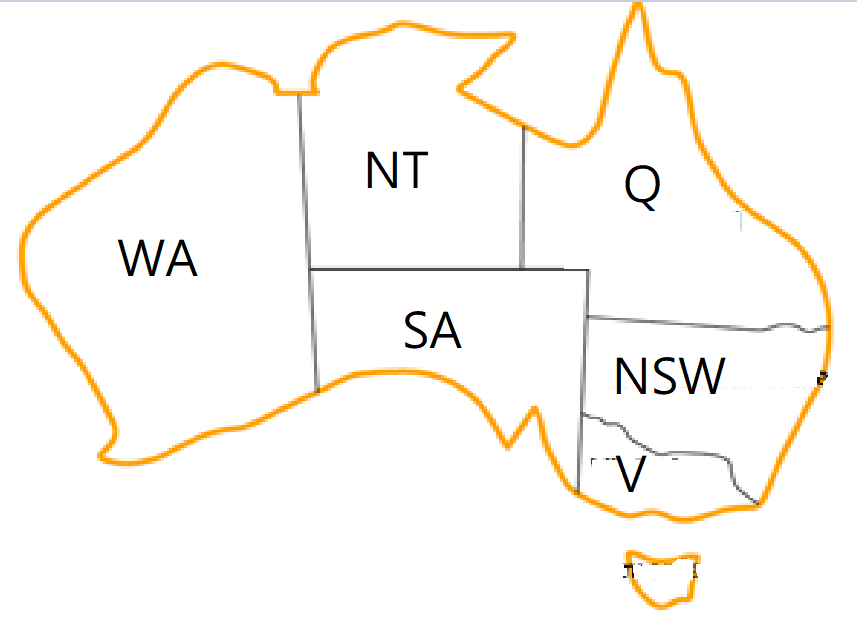
1. Defining a list of allowed values for the variables (domain)



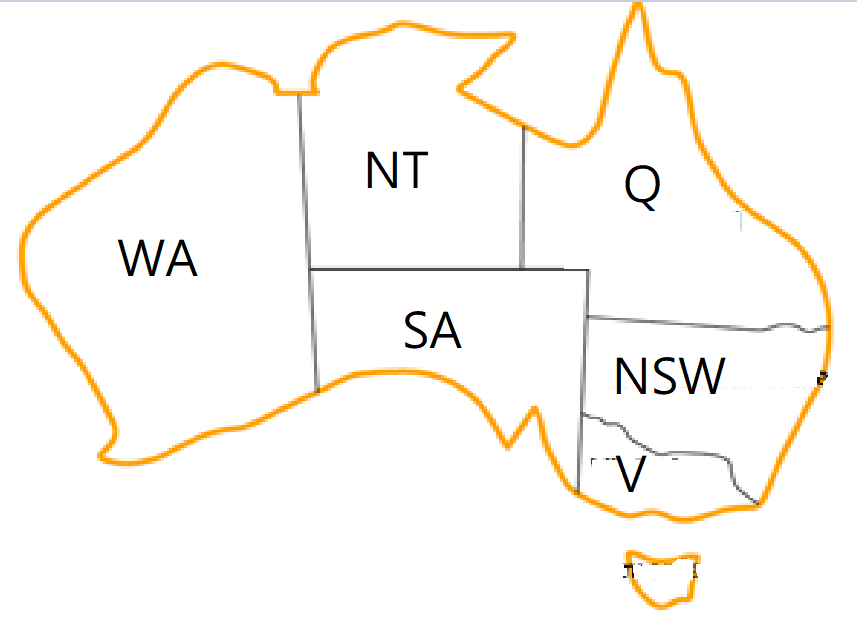
1. Define the functions for checking the Constraints

* We will write a function that return **True** if the constraints are obeyed (so far)
* Variable with None are ignored for the time being (they will be checked when assigned)
* This function differs from problem to problem
* For map colouring, we need to make sure each two neighbouring regions have different colours

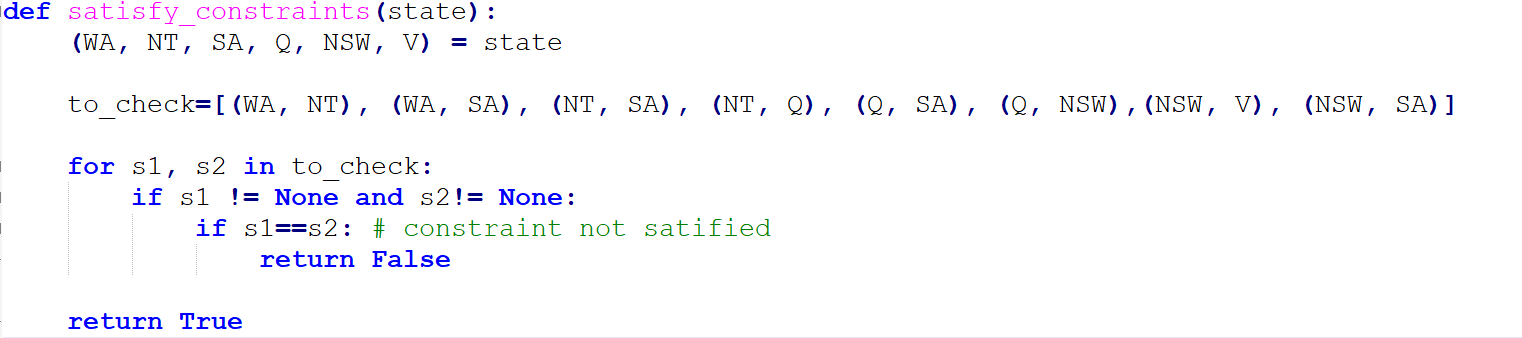




3. Functions for checking the Constraints

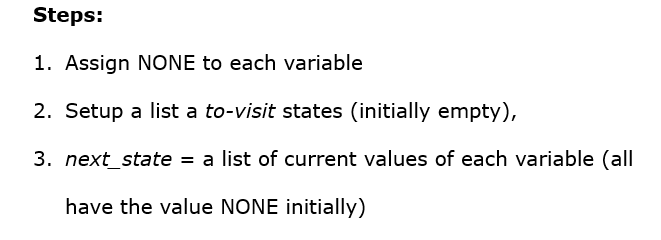
* We will write a function that return **True** if the constraints are obeyed (so far)
* Variable with None are ignored for the time being (they will be checked when assigned)
* This function differs from problem to problem
* For map colouring, we need to make sure each two neighbouring regions have different colours

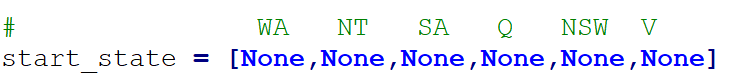
**Alternative Version:**

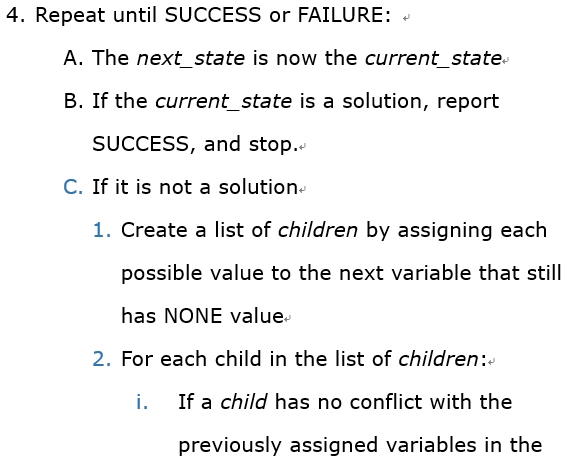


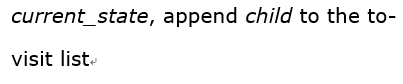
**Example: Map Colouring**

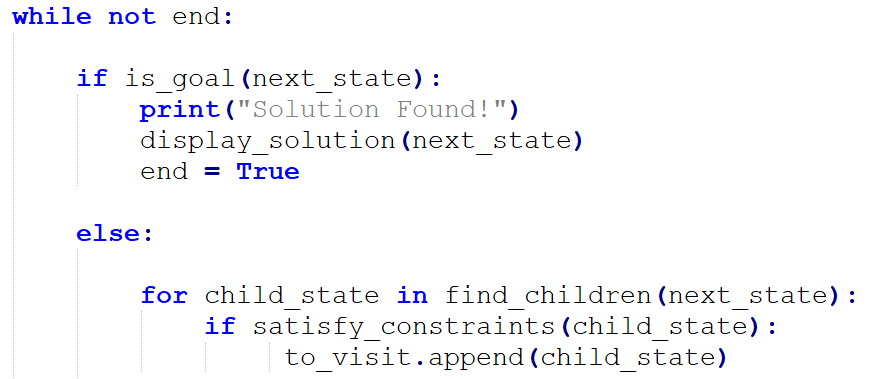
**The main procedure (common for most problems)**

****

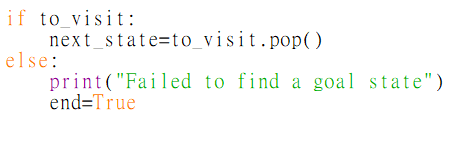
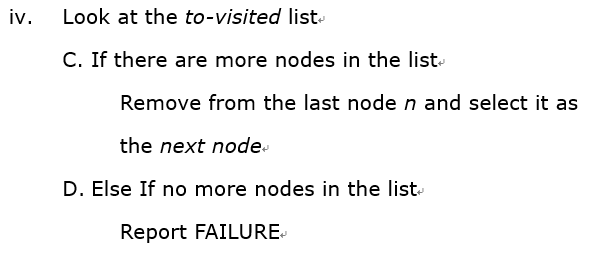
****

**Example: Map Colouring**

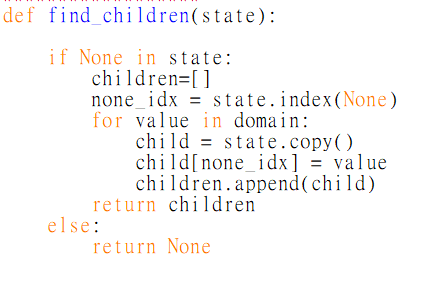




**Note: *Satisfy\_constraints*** is a function that verify whether the variable assigments (so far) follow the required constraints or not (See Below)

 **Example: Map Colouring**

**Example: Map Colouring**



The find\_children(state) function:

1. Find the first variable that has None as its value (i.e., not yet assigned)
2. Create new states by replacing it in turn with each allowed values.
3. Return all the new states in a list

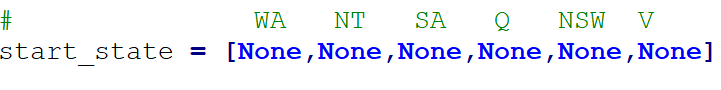
["R", "B", **None**, None, ….]

[["R", "B", **"R"**, None, ….] ["R", "B", **"G"**, None, ….] ["R", "B", **"B"**, None, ….]]

(any conflicts will be dealt with in the next step)

**Example: Map Colouring: Summary**

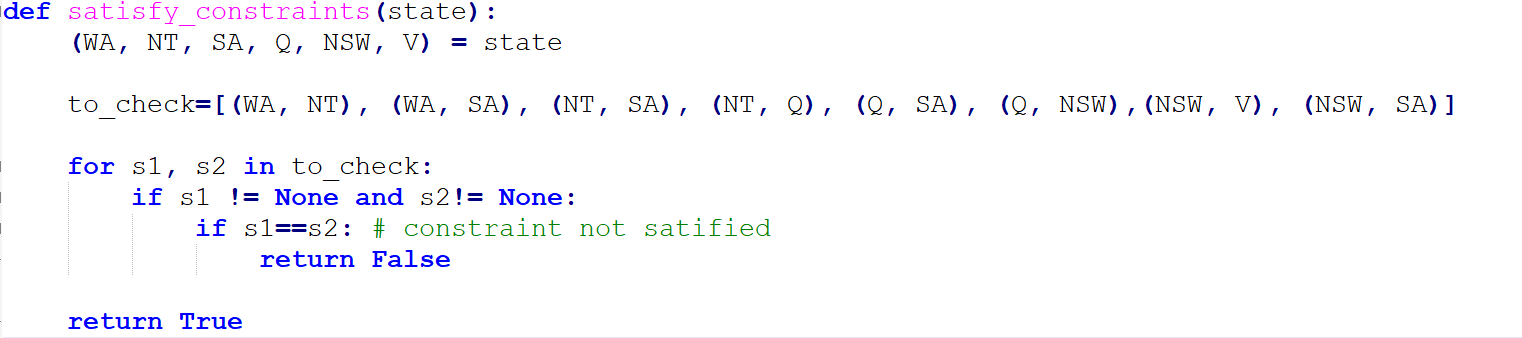
**Variables**



**Domains**



**Constraints**

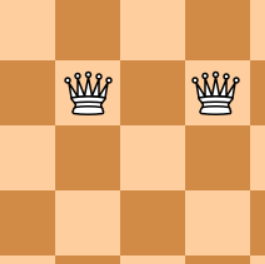
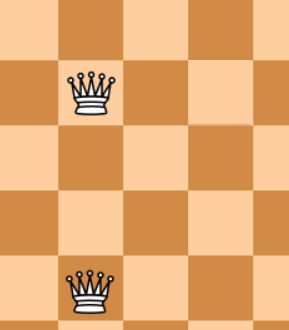
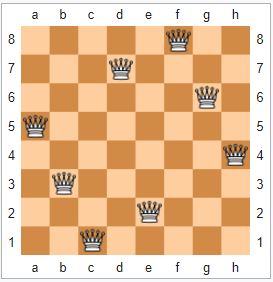


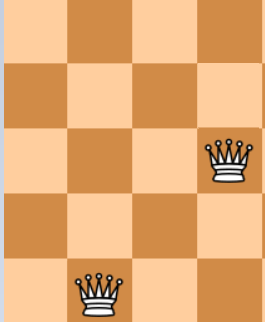
**Complete programming source code listing: See Moodle**

**Example: 8 Queens Problem**

**Problem:**

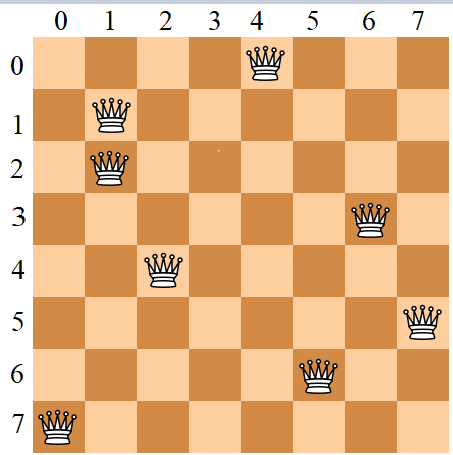
Given a 8\*8 chess board, place 8 *Queens* on it, such that no two queens share the same row, column, or diagonals

****

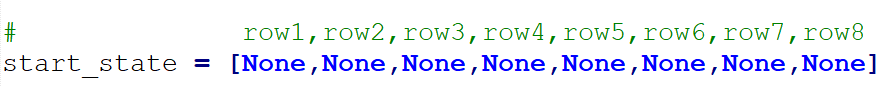
****

**Solution:**

1. Defining a list of **variables**

* We first realize that *there must be exact one queen in each row*
* Therefore we can represent a state by the position of the queen in each row.
* ****For example for board in the right can be represented as a state of 8 variables as

state = q[4,1,1,6,4,7,5,0]

* And the start\_state is defined as

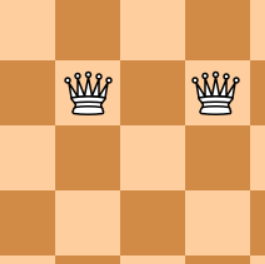
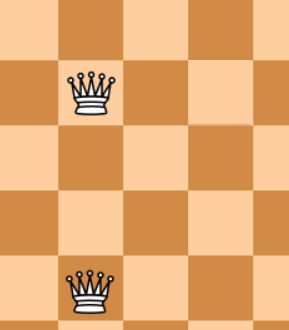
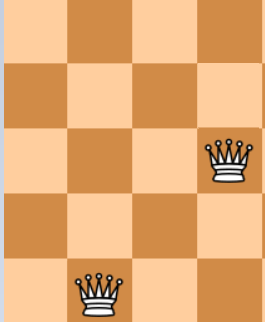
1. Defining a list of allowed values for the variables (the **domains**)

* The domain is from 0 to 7, i.e.,



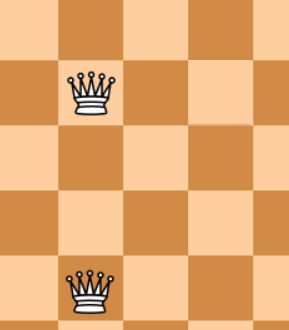
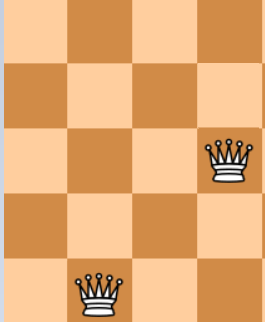
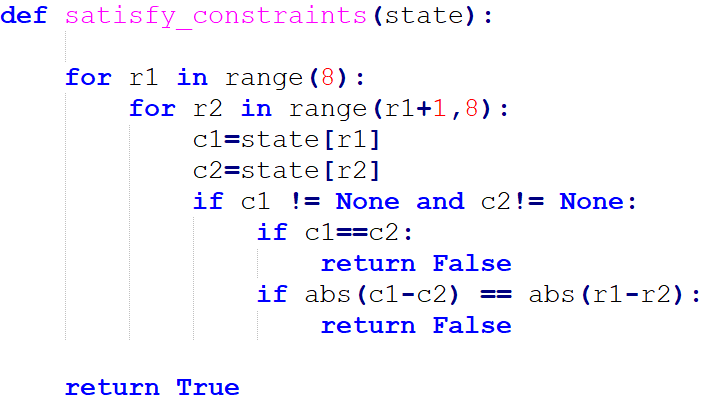
1. Define the functions for checking the **Constraints**

There are three constraints to check:

1. No two queens can be on the same row.
   * However, this is already handled by our chosen representation.
2. No two queens can be on the same column
   * This means none of the 8 variables can have the same value
3. No two queens can be on the same diagonal.
   * Suppose two queens are in (row1, col1) and (row2, col2),

This means abs(row1-row2) != abs(col1-col2)

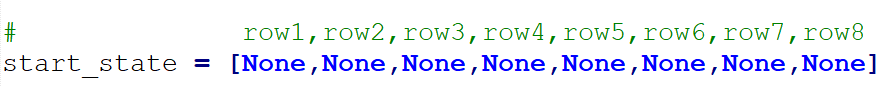
3. Define the functions for checking the Constraints

****The codes are as follow:

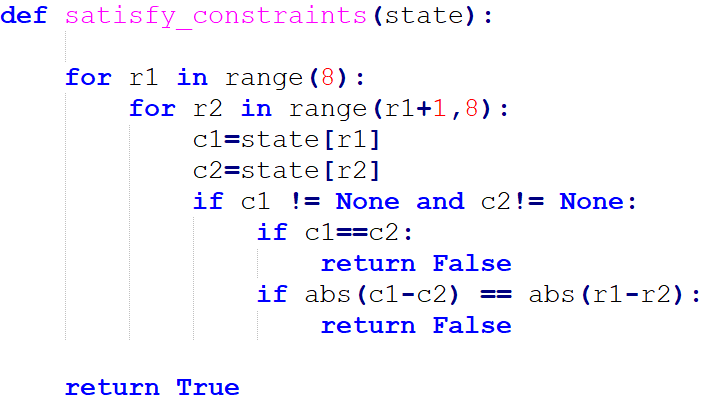
We look at each pair of two rows in turn

**Example: 8 Queens Problem: Summary**

The complete program code is on Moodle.

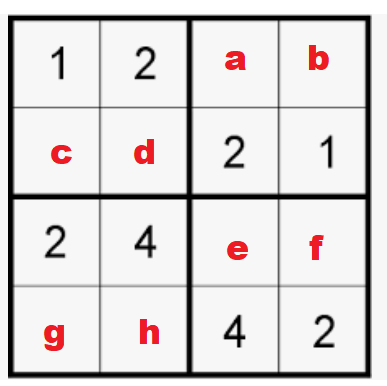
**Variables**

**Domain**

**Constraints**

**Exercise 2: 4 \* 4 Sudoku**

Write a program to solve the 4\*4 Sudoku problem below:

**Variables:** a,b,c,d,e,f,g,h

**Domain:** {1,2,3,4}

**Constraints:**

* Each small squares must contains 1 – 4:
* For example, constraint for the upper left squares: sorted[1,2,c,d] == [1,2,3,4]
* Each row/column must contain 1 – 4

**Notes:** can we add extra constraints to speed up the search.  
For example, if the state is [1, none, none, none, none, none, none, none]

(i.e., a is 1), does it violate the constraint already? Do we really need to search its children?

**REFERENCE / SELF STUDY**

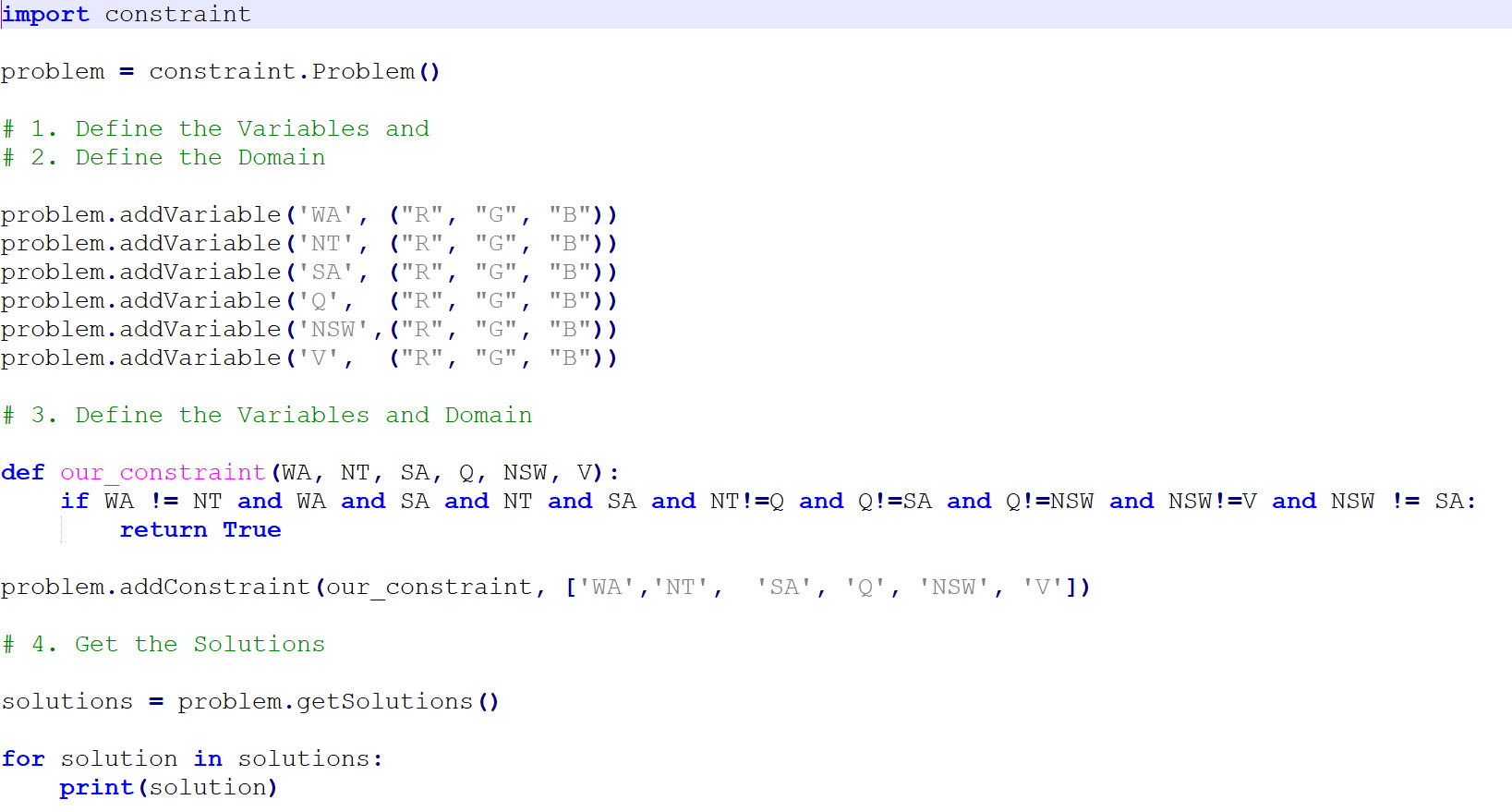
**Python Contraint Library**

* Python has a ***python-constraint***library, whose function is similar to the one we defined.
* To install in IDLE, go to your command prompt (with admin right) and type

pip install python-constraint

**REFERENCE / SELF STUDY**

**Example: Australia Problem with *Python-Contraint* Library**

****

Domains

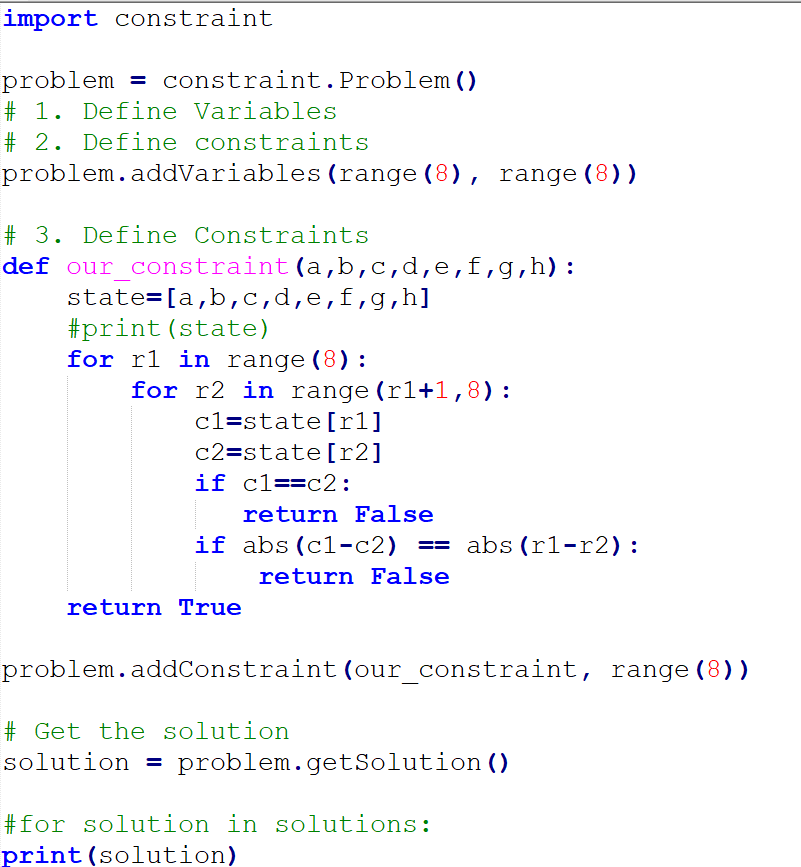
Variables

Constraint Function

Call *our\_constraint*() function using those variables (WA, NT etc), with combinations of possible values in their domains

**REFERENCE / SELF STUDY**

**Example: 8 Queens Problem with Python-Contraint Library**

****

Define 8 Unnamed Variables: var0, var1, … var 7

Domain (0,1,2,3,4,5,6,7)

Call *our\_constraint*() function using the eight variables (var0 … var7), with combinations of possible values in the domains

Constraint Function (Very similar to the one before)