Fundamentals of Programming 2

Backtracking Algorithms

Arkadiusz Chrobot

Department of Computer Science

June 1, 2020

Outline

Introduction

Backtracking algorithms are often used for solving problems presented as a riddle where the input data and the goal or at least its characteristics are given, but the way of achieving this goal is unknown. Examples of such issues are chess problems like the eight queens problem or the knight's tour.
There is no specific and efficient way of solving such problems. Usually the only feasible approach is using the “trial and error” strategy. Since it is a tedious task it is beneficial to apply a computer for it by adjusting and implementing a backtracking algorithm for a particular problem.

The Water Jug Problem

The application of the backtracking algorithm can be demonstrated with a quite simple problem named “the water jug problem”:

Definition

The Water Jug Problem: With a four litre jug and a three litre jug measure exactly two litres of water. The problem is solved when any of the jugs contains the desired amount of water.

The computer is not necessary for solving such a problem. Even using a pen and a piece of paper should be enough. Nevertheless, its simplicity is an advantage — it is easier to apply the backtracking algorithm for solving it.

Analysis of the Problem

Let's take a closer look at the problem. There are available two jugs and an unlimited source of water. The task is to measure two litres of water, by filling the jugs, pouring the water from one into the other or emptying them. Thus, the number of actions that can be performed on the jugs is limited. Moreover, in one step only one action can be taken. A more detailed analysis shows that each of these actions results in leaving in the jugs a discrete amount of water, i.e. one that can be expressed with the use of natural numbers. Hence, the amount of the water in both jugs, or the *state* of the jugs can be described with a pair of such numbers. Many such states can be found while trying to solve the problem. They form a *discrete solution space*. Some of the states are the ones that are sought after — where one of the jugs contains two litres of water — they satisfy the *goal condition*. If the definition of the problem stated that the desired state is only one, where, for example, the four litre jug contains two litres of water, then the state would be a single *goal*.

Analysis of the Problem

The transition from one state to another is possible only when an *operation* (an action like filling, emptying or pouring) is performed that changes the amount of water in jugs. However, not all of the operations can be applied in each possible state. For example it is impossible to pour water out of an empty jug. A state has to meet a specific condition that allows the operation to be performed. In the next two slides is presented a list of *operators* that are a formal notation for describing all possible operations which can be performed on jugs. The "$\rightarrow$" symbol means a transition from one state of jugs to another, caused by the operation. The `j3` and `j4` variables denote the amount of water in the three litre and four litre jug respectively. The expression on the right side of the "$\rightarrow$" symbol specifies the condition that has to be satisfied by the initial state, so that the operation described by the operator could be performed. The expression on the left side of the symbol describes the state of the jugs after the operation is done (the next state).

Analysis of the Problem

Operators

1: Fill the four litre jug

$(j4, j3 | j4 < 4) \longrightarrow (4, j3)$

2: Fill the three litre jug

$(j4, j3 | j3 < 3) \longrightarrow (j4, 3)$

3: Empty the four litre jug

$(j4, j3 | j4 > 0) \longrightarrow (0, j3)$

4: Empty the three litre jug

$(j4, j3 | j3 > 0) \longrightarrow (j4, 0)$

Analysis of the Problem

Operators

5: Pour the water from the three litre jug into the four litre jug, to fill the latter

$(j4, j3 | j4 + j3 \geq 4 \wedge j3 > 0) \rightarrow (4, j3 - (4 - j4))$

6: Pour the water from the four litre jug into the three litre jug, to fill the latter

$(j4, j3 | j4 + j3 \geq 3 \wedge j4 > 0) \rightarrow (j4 - (3 - j3), 3)$

7: Pour the water from the three litre jug into the four litre jug, to empty the former

$(j4, j3 | j4 + j3 \leq 4 \wedge j3 > 0) \rightarrow (j3 + j4, 0)$

8: Pour the water from the four litre jug into the three litre jug, to empty the former

$(j4, j3 | j4 + j3 \leq 3 \wedge j4 > 0) \rightarrow (0, j3 + j4)$

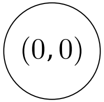
The Algorithm

As it has been mentioned before there is a space of states that describe the amount of the water in the jugs. Transitions between those states are possible with the use of operations defined by the operators. This space can be expressed as a directed graph where the vertices are the states of jugs and the edges are operations that make it possible to leave one state and enter another. So, finding the solution of the water jug problem reduces to finding a path in the graph leading from the initial vertex that describes the state where the jugs are empty to one of the *goal vertices* specifying the state where one of the jugs contains exactly two litres of water. To find such a path the DFS algorithm can be applied. There is however one issue left — the graph is mostly unknown. Only the initial vertex and the set of operators that can form the edges of the graph are known. However, this information is all that is needed to create the whole graph. On the other hand, building the whole graph is unnecessary.

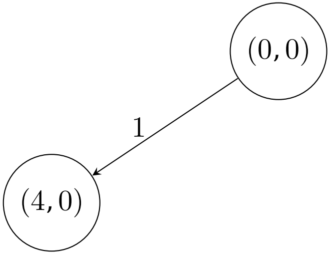
The Algorithm

It is enough to generate the vertices that belong to the currently explored path in the graph. If a newly created vertex repeats then it is necessary to *backtrack* to the previous vertex and try to create another one (a different state) that also belongs to the path. Making new vertices should be stopped after a vertex that corresponds to one of the states that satisfy the goal condition is created. The resulting path is the solution of the water jug problem. This is the essence of backtracking algorithm. The next slide shows an animation that (partially) illustrates how this algorithm works.

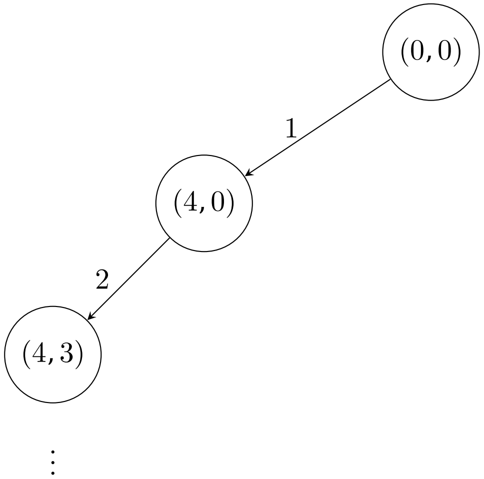
The Algorithm



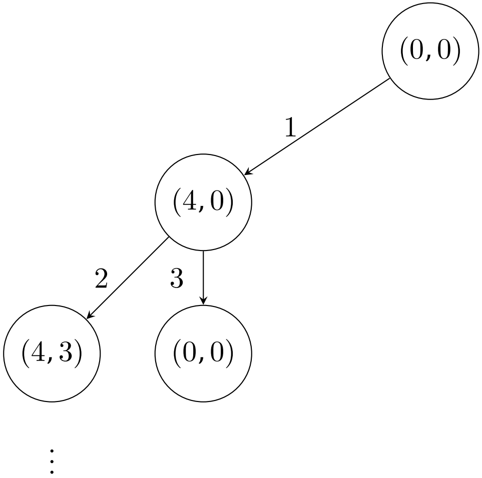
The Algorithm



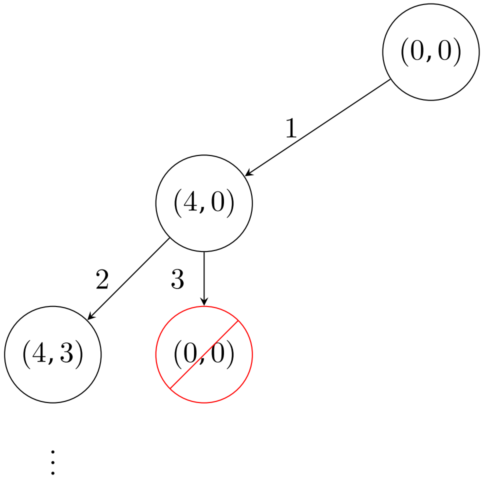
The Algorithm



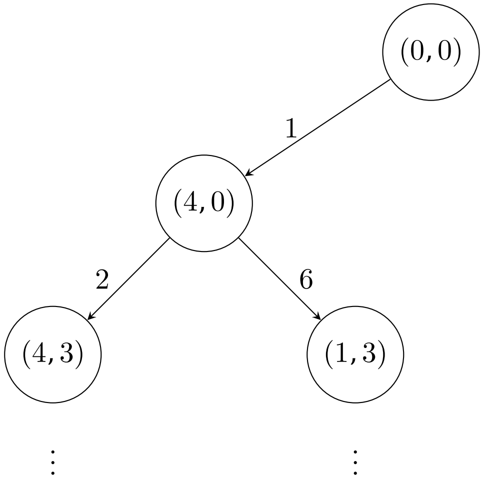
The Algorithm



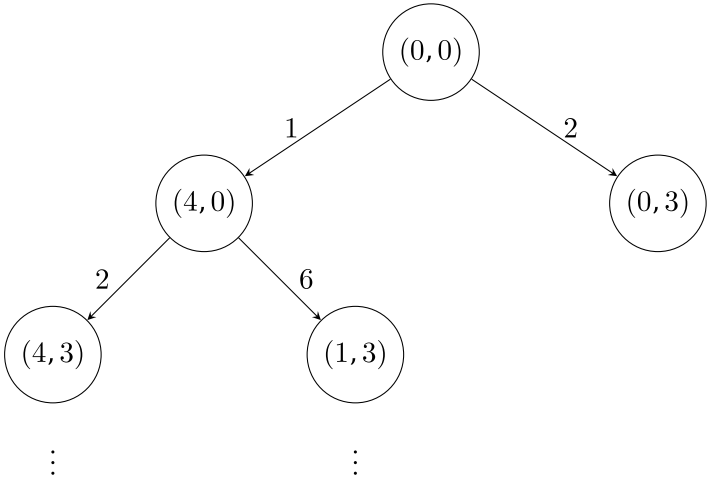
The Algorithm



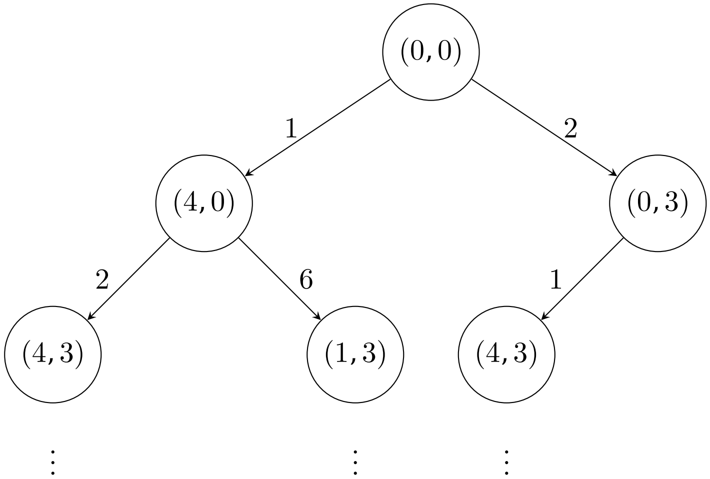
The Algorithm



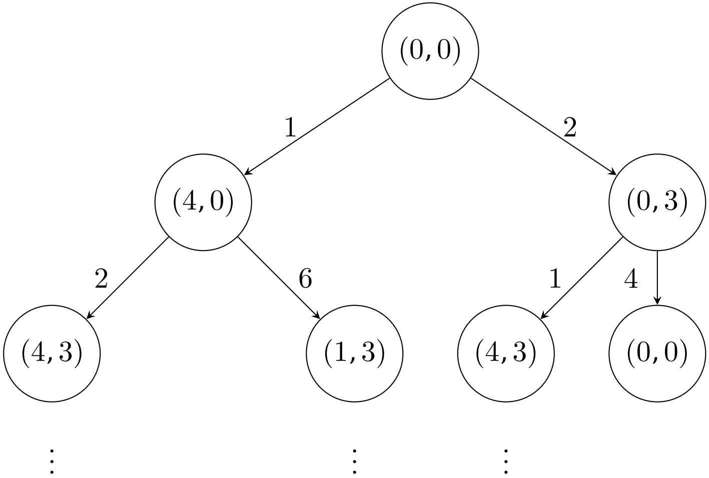
The Algorithm



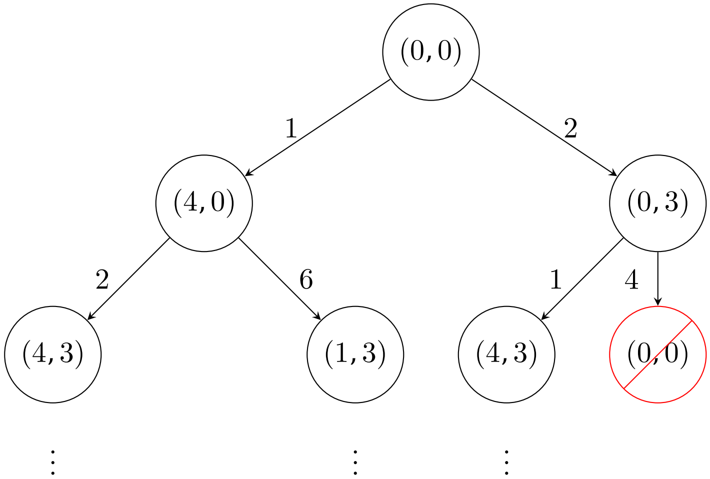
The Algorithm



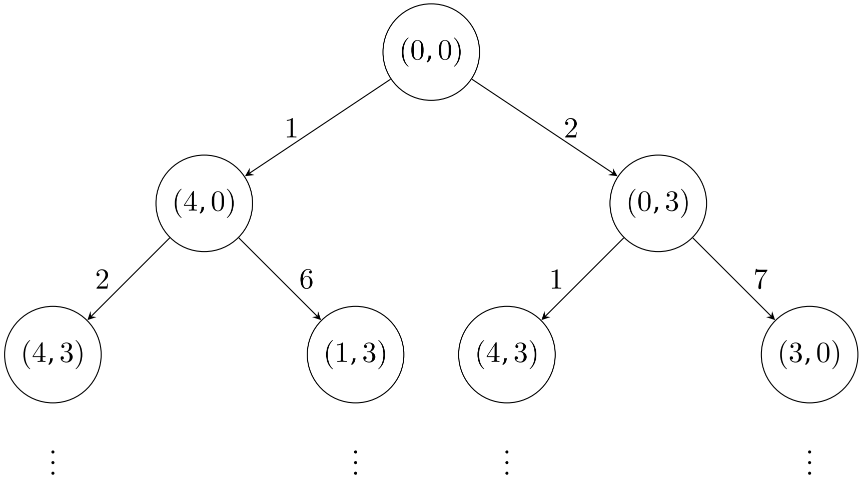
The Algorithm



The Algorithm



The Algorithm



The Algorithm
The Single Solution Version

To make the implementation of the algorithm easier it is first expressed in a pseudocode which is an intermediate form between a natural language and the computer code. The version of the algorithm presented on the next slide finds only one solution of the water jug problem.

The Algorithm
The Single Solution Version

The Pseudocode for the Single Solution Version

```
find_solution(path)
{
    for each(operator){
        if(can_apply(operator, last_state(path))){
            new_state = operator(last_state(path));
            if(has_not_been(path, new_state)){
                add(path, new_state);
                if(goal_condition(new_state))
                    print(path);
                else
                    find_solution(path);
            } else
                remove(new_state);
        }
    }
}
```

The Algorithm
The Many Solutions Version

It can be deduced from the definition that the water jug problem has more than one solution. The question arises how to modify the algorithm from the previous slide, so that it can find all possible solutions to the problem. To this end the concept of “backtracking” has to be expanded. The algorithm should go back to the previously created vertex not only when the new one repeats in the path, but always when the exploration of one of the paths associated with the previous vertex is finished. In that way the algorithm can explore all paths associated with each of the vertices, what allows it to find all possible solutions to the problem. It means that the pseudocode from the previous slide has to be supplemented with one additional statement that removes the last vertex from the path when the algorithm returns from a recursive call or prints the solution (the path). The modification is shown in the pseudocode presented in the next slide.

The Algorithm

The Many Solutions Version

The Pseudocode for The Many Solutions Version

```
find_solution(path)
{
    for each(operator){
        if(can_apply(operator, last_state(path))){
            new_state = operator(last_state(path));
            if(has_not_been(path, new_state)){
                add(path, new_state);
                if(goal_condition(new_state))
                    print(path);
                else
                    find_solution(path);
                remove_last_state(path);
            } else
                remove(new_state);
        }
    }
}
```

The Backtracking Algorithm — Implementation

The source code of a program that implements the backtracking algorithm finding all solutions to the water jug problem is presented in the next slides. It is a quite complex program, but each part of it is commented to make it easier to understand.

The Backtracking Algorithm — Implementation

Used Header Files and Definitions

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define NUMBER_OF_OPERATORS 8

enum operator_index {NONE = -1, FILL_4_LITRE, FILL_3_LITRE,
        EMPTY_4_LITRE, EMPTY_3_LITRE, POUR_3_FILL_4,
        POUR_4_FILL_3, POUR_AND_EMPTY_4, POUR_AND_EMPTY_3};

struct jugs_states {
    unsigned int jug_3_litre_state, jug_4_litre_state;
};
```

The Backtracking Algorithm — Implementation

Used Header Files and Definitions

Lines 1–3 of the code from the previous slide contain directives that include header files to the program. A constant that specifies the number of the operators used for solving the water jug problem is defined in the 5th line. Because the operators are stored in an array, the constant also defines the number of elements of that array. The lines 7–9 contain a definition of an enumerated type. A variable of this type is used for indexing the array of operators that also create vertices of the graph. The operators are numerated starting from zero, but the first element of the enumerated type has a value of `-1`. It is used for marking that no operator has been used for creating a vertex that specifies the initial state. The lines 11–12 contain definition of a structure type, which fields are used for storing the information about the amount of water in both of the jugs.

The Backtracking Algorithm — Implementation

Definitions of Data Types

```c
struct queue_node {
    struct jugs_states states;
    enum operator_index operator_number;
    struct queue_node *next;
};

struct queue_pointers {
    struct queue_node *head, *tail;
} queue;

typedef bool (*operator_condtion_function_pointer)
                                    (struct jugs_states state);

typedef struct jugs_states(*operator_function_pointer)
                                    (struct jugs_states state);
```

The Backtracking Algorithm — Implementation

Definitions of Data Types

Lines 1–5 of the code from the previous slide contain a definition of the base type of a queue used for storing the currently explored path of the graph. It is an input-restricted double-ended queue. The `state` field in a structure of this type is used for storing information about the state of jugs, the `operator_number` field stores the number of the operator used for creating the state. The `next` field is a pointer to a next element of the queue. The type of pointers structure for this queue is defined in lines 7–9 along with a variable of this type (9th line). Another type is defined in lines 11–12. It is a function pointer type. A pointer of this type can point to a function that takes as an argument a structure that describes a state of jugs and returns a `bool` value that informs if an operator can be applied to this state. In other words this function evaluates the condition on the left side of the "$\rightarrow$" symbol in the formal definition of the operator.

The Backtracking Algorithm — Implementation

Definitions of Data Types

Lines 14–15 contain a definition of another function pointer type. This time the pointer of that type can point to a function that creates a new state of jugs using its argument, which is their previous state. It means that the function corresponds to the part of the formal definition of an operator that is located on the right side of the "$\rightarrow$" symbol.

The next five slides present definitions of functions that can be pointed by pointers of this two pointer types. Please notice, that their code corresponds to the formal definitions of operators presented in the earlier slides.

The Backtracking Algorithm — Implementation

The Operators Functions

```c
bool can_fill_4_litre_jug(struct jugs_states state)
{
    return state.jug_4_litre_state<4;
}

struct jugs_states fill_4_litre_jug(struct jugs_states state)
{
    state.jug_4_litre_state = 4;
    return state;
}

bool can_fill_3_litre_jug(struct jugs_states state)
{
    return state.jug_3_litre_state<3;
}
```

The Backtracking Algorithm — Implementation

The Operators Functions

```c
struct jugs_states fill_3_litre_jug(struct jugs_states state)
{
    state.jug_3_litre_state = 3;
    return state;
}

bool can_empty_4_litre_jug(struct jugs_states state)
{
    return state.jug_4_litre_state>0;
}

struct jugs_states empty_4_litre_jug(struct jugs_states state)
{
    state.jug_4_litre_state = 0;
    return state;
}
```

The Backtracking Algorithm — Implementation

The Operators Functions

```
bool can_empty_3_litre_jug(struct jugs_states state)
{
    return state.jug_3_litre_state>0;
}

struct jugs_states empty_3_litre_jug(struct jugs_states state)
{
    state.jug_3_litre_state = 0;
    return state;
}

bool can_fill_up_4_litre_jug_with_3_litre
                                   (struct jugs_states state)
{
    return state.jug_4_litre_state + state.jug_3_litre_state
                            >= 4 && state.jug_3_litre_state>0;
}
```

The Backtracking Algorithm — Implementation

The Operators Functions

```
struct jugs_states empty_3_litre_jug_to_4_litre
                                                 (struct jugs_states state)
{
     state.jug_4_litre_state = state.jug_3_litre_state +
                                                     state.jug_4_litre_state;
     state.jug_3_litre_state = 0;
     return state;
}

bool can_empty_4_litre_jug_to_3_litre(struct jugs_states state)
{
     return state.jug_3_litre_state +
          state.jug_4_litre_state <= 3 &&
          state.jug_4_litre_state > 0;
}
```

The Backtracking Algorithm — Implementation

The Operators Functions

```c
struct jugs_states empty_4_litre_jug_to_3_litre
                                            (struct jugs_states state)
{
    state.jug_3_litre_state = state.jug_3_litre_state +
                                             state.jug_4_litre_state;
    state.jug_4_litre_state = 0;
    return state;
}
```

The Backtracking Algorithm — Implementation

The Operators Array

```c
struct operator_structure {
    operator_condtion_function_pointer is_condition_fullfiled;
    operator_function_pointer get_next_state;
} operators[NUMBER_OF_OPERATORS] = {
    [FILL_4_LITRE] = {
        .is_condition_fullfiled = can_fill_4_litre_jug,
        .get_next_state = fill_4_litre_jug
    },
    [FILL_3_LITRE] = {
        .is_condition_fullfiled = can_fill_3_litre_jug,
        .get_next_state = fill_3_litre_jug
    },
    [EMPTY_4_LITRE] = {
        .is_condition_fullfiled = can_empty_4_litre_jug,
        .get_next_state = empty_4_litre_jug
    },
```

The Backtracking Algorithm — Implementation

The Operators Array

```
        [EMPTY_3_LITRE] = {
            .is_condition_fullfiled = can_empty_3_litre_jug,
            .get_next_state = empty_3_litre_jug
        },
        [POUR_3_FILL_4] = {
            .is_condition_fullfiled = can_fill_up_4_litre_jug_with_3_litre,
            .get_next_state = fill_up_4_litre_jug_with_3_litre
        },
        [POUR_4_FILL_3] = {
            .is_condition_fullfiled = can_fill_up_3_litre_jug_with_4_litre,
            .get_next_state = fill_up_3_litre_jug_with_4_litre
        },
        [POUR_AND_EMPTY_4] = {
            .is_condition_fullfiled = can_empty_4_litre_jug_to_3_litre,
            .get_next_state = empty_4_litre_jug_to_3_litre
        },
```

The Backtracking Algorithm — Implementation

The Operators Array

```
        [POUR_AND_EMPTY_3] = {
            .is_condition_fullfiled =
                                    can_empty_3_litre_jug_to_4_litre,
            .get_next_state = empty_3_litre_jug_to_4_litre
        }
};
```

The Backtracking Algorithm — Implementation

The Operators Array

Three last slides presents declaration and initialization of the operators array. It is an array of elements that are structures of pointers to a function. A concept from the object oriented programming is applied here — the pointed functions can be regarded as methods of an object. The type of elements of this array is defined in lines 1–4 in the first of the mentioned slides. It specifies a structure which both fields are pointers to a function. Types of that pointers have been defined in the 19th slide. The array is declared in the 27th slide, in the 4th line. The rest of lines, also in the other slides, initializes elements of the array. To this end a *designated initializer* is applied, which allows the programmer to assign a value to a specific element of the array, by placing its index in brackets and using an assignment operator. For example the 6th element of an array of ten elements of the `int` type can be initialized using the designated initializer as follows:

```
int array[10] = {[5]=7};
```

The Backtracking Algorithm — Implementation

The Operators Array

The rest of the elements of the example array gets the value of `0`. For initializing the elements of the operators array the elements of the `operator_index` enumerated type are used as indices. Because the operators array is an array of pointers structures, an address of a function associated with a specific operator is assigned to each field of each element.

The Backtracking Algorithm — Implementation

The `enqueue()` and `dequeue()` Functions

```c
void enqueue(struct queue_pointers *queue,
                                        struct queue_node *new_node)
{
    queue->tail->next = new_node;
    queue->tail = new_node;
}

void dequeue(struct queue_pointers *queue)
{
    if(queue->head) {
        struct queue_node *tmp = queue->head->next;
        free(queue->head);
        queue->head=tmp;
        if(tmp==NULL)
            queue->tail = NULL;
    }
}
```

The Backtracking Algorithm — Implementation

The `enqueue()` and `dequeue()` Functions

The previous slide presents definitions of two functions. The first one adds a new element and the second one removes the first element from a queue that stores the currently explored path in the graph. The `enqueue()` function, defined in the lines 1–6, doesn't return any value, but takes two arguments: an address of the queue pointers structure and an address of the new element that it adds at the tail of the queue. The function assumes that the queue is not empty. i.e. it has at least one element. In the 4th line the function assigns the address of the new element to the `next` field of the last element of the queue, then it assigns the same address to the pointer to the last element of the queue (5th line) and exits. The `dequeue()` function removes an element at the head of the queue. It is defined in the same way as in the program from the previous lecture that demonstrates the DFS algorithm.

The Backtracking Algorithm — Implementation

The `remove_queue()` Function

```c
void remove_queue(struct queue_pointers *queue)
{
    while(queue->head)
        dequeue(queue);
}
```

The Backtracking Algorithm — Implementation

The `remove_queue()` Function

The `remove_queue()` function deletes the whole queue and it is defined in the same way as in the previous lecture.

The Backtracking Algorithm — Implementation

The `has_already_been()` Function

```c
bool has_already_been(struct queue_pointers queue,
                                        struct queue_node *new_node)
{
    while(queue.head) {
        if(queue.head->states.jug_4_litre_state ==
                new_node->states.jug_4_litre_state &&
                queue.head->states.jug_3_litre_state ==
                         new_node->states.jug_3_litre_state)
            return true;
        queue.head = queue.head->next;
    }
    return false;
}
```

The Backtracking Algorithm — Implementation

The `has_already_been()` Function

The `has_already_been()` function verifies if the new element of the queue specifies a state (a vertex of the graph) that has been already found. The function returns a value of the `bool` type. If it is `true` then it means that the state specified by the new element has been already processed. If it is `false` then this state have not been discovered yet. In the `while` loop the function traverses the queue and compares the state stored in its elements with the state stored in the new element (lines 5–8). If one of the elements in the queue stores the same state as the new one, the function returns `true` and exits (line no. 9). If none of the elements stores the same state as the new one, then the `while` loop stops after traversing the queue and the function returns `false` (12th line).

The Backtracking Algorithm — Implementation

The `remove_tail()` Function

```c
void remove_tail(struct queue_pointers *queue)
{
    if(queue->head) {
        if(queue->head == queue->tail) {
            free(queue->head);
            queue->head = queue->tail = NULL;
            return;
        }
        struct queue_node *node = queue->head;
        while(node->next!=queue->tail)
            node = node->next;
        free(queue->tail);
        node->next = NULL;
        queue->tail = node;
    }
}
```

The Backtracking Algorithm — Implementation

The `remove_tail()` Function

The `remove_tail()` function removes the last element from the queue to allow the backtracking algorithm to find a new solutions of the water jug problem. The `remove_tail()` function takes the address of the queue pointers structure as its only argument and does not return any value. In the 4th line, by comparing the values of the `head` and the `tail` pointers it checks, if the queue has only one element. If so, it deletes that element (5th line), assigns the `NULL` value to both queue pointers (6th line) and exits (7th line). If the queue has more than one element then the function looks for the last but one element of the queue in the `while` loop (lines 10–11). This element stores in its `next` field an address of the last element of the queue. After it is found the function removes the last element from the queue (12th line), assigns the `NULL` value to the `next` field of the new last element of the queue (13th line) and stores the address of that element in the `tail` pointer of the queue (14th line).

The Backtracking Algorithm — Implementation

The `print_solution()` Function

```c
void print_solution(struct queue_pointers queue)
{
    static unsigned char solution_number;
    unsigned char step = 1;
    char * operators_description[NUMBER_OF_OPERATORS] = {
        "Fill the 4 litre jug.",
        "Fill the 3 litre jug.",
        "Empty the 4 litre jug.",
        "Empty the 3 litre jug.",
        "Pour the water from the 3 litre jug into the 4 litre jug,\
 to fill the latter.",
        "Pour the water from the 4 litre jug into the 3 litre jug,\
 to fill the latter.",
        "Pour the water from the 3 litre jug into the 4 litre jug,\
 to empty the former.",
        "Pour the water from the 4 litre jug into the 3 litre jug,\
 to empty the former."
    };
```

The Backtracking Algorithm — Implementation

The print_solution() Function

```
    printf("Solution no. %hhu:\n",++solution_number);
    while(queue.head) {
        enum operator_index operator =
                queue.head->operator_number;
        if(operator!=NONE) {
            printf("Step number %hhu:\n",step++);
            printf("%s\n",operators_description[operator]);
        } else
            puts("Initial state:");
        printf("Water level in the 4 litre jug: %u and 3 litre\
        jug: %u\n",
                queue.head->states.jug_4_litre_state,
                queue.head->states.jug_3_litre_state);
        getchar();
        queue.head = queue.head->next;
    }
    puts("THE END");
}
```

The Backtracking Algorithm — Implementation

The `print_solution()` Function

The `print_solution()` function displays details of one of the water jug problem solutions found by the program. This task mainly consists in interpreting the data about the path stored in the queue. The function takes the structure of queue pointers as an argument and returns no value. A static local variable for numbering the solutions printed by the function is declared in the 3rd line. All static variables are initialized by default with the `0` and are not destroyed between subsequent function calls. Another variable is declared in the 4th line. This one is used for numbering the steps of a single solution (subsequent actions) and it is an ordinary local variable, initiated with the value of `1`. An array of strings that describe in English the actions performed by the operators is declared in lines 5–18.

The Backtracking Algorithm — Implementation

The `print_solution()` Function

In the 1st line of the second slide with the `print_solution()` function definition a message is printed that informs which solution this function will currently display. The number of the solution is calculated by using the pre-increment operator on the `solution_number` variable. In the `while` loop the function traverses the queue and assigns the operator number stored in each element to the `operator` variable and then compares it with the value of the `NONE` element of the `operator_index` enumerated type. If they are not equal then the function increments the value of the `step` variable by one and displays it together with the description of the operator's action (6th line). Otherwise the function prints a message informing that it is displaying information about the initial state. The next actions performed by the function are the same for all the states.

The Backtracking Algorithm — Implementation

The `print_solution()` Function

The state of the water in jugs that is stored in the queue element currently visited by the loop is displayed in the lines 10–13 of the code from the second slide with the definition of the `print_solution()` function. Next, the function waits for the user to press the Enter key on the keyboard (14th line). After that the loop visits the next element of the queue (15 th line). When the loop stops, the function displays a message that informs the user that it has finished printing a single solution of the water jug problem and exits.

The Backtracking Algorithm — Implementation

The create_new_state() Function

```c
struct queue_node *create_new_state(struct jugs_states state,
                                    enum operator_index operator_number)
{
    struct queue_node *new_node = (struct queue_node *)
                                    malloc(sizeof(struct queue_node));
    if(new_node) {
        new_node->states = state;
        new_node->operator_number = operator_number;
        new_node->next = NULL;
    }
    return new_node;
}
```

The Backtracking Algorithm — Implementation

The `create_new_state()` Function

The `create_new_state()` function creates a new element of the queue that specifies a newly generated state of water in jugs. It takes as arguments the structure that describes the new state and the number of the operator that has been used for creating this state. The function returns an address of the new queue element that stores both those data items. It allocates memory for the new queue element in lines 4–5. If the allocation is successful then the condition in the 6th line is met and the function initializes the fields of the new element. It assigns the state of water in jugs to the field of the element in the 7th line. The number of the operator is assigned to another field in the 8th line. Finally, the `NULL` value is assigned to the element `next` field in the 9th line. The function returns the address of the new element of the queue or the `NULL` value, if it has been unable to create such an element, and exits (11th line).

The Backtracking Algorithm — Implementation

The initialize_queue() Function

```c
void initialize_queue(struct queue_pointers *queue)
{
    struct queue_node *first_state = (struct queue_node *)
                                 malloc(sizeof(struct queue_node));
    if(first_state) {
        first_state->states.jug_4_litre_state =
                    first_state->states.jug_3_litre_state = 0;
        first_state->operator_number = NONE;
        first_state->next = NULL;
        queue->head = queue->tail = first_state;
    }
}
```

The Backtracking Algorithm — Implementation

The `initialize_queue()` Function

The `initialize_queue()` function initializes the queue by adding its first element which describes the initial vertex of the path that in turn describes a state in which both jugs are empty. Because the function is invoked before the `enqueue()` function, the latter can skip verifying if the queue has at least one element. The `initialize_queue()` function takes as an argument the address of the queue pointers structure and returns no value. It allocates memory for the first element of the queue in lines 3–4. If the allocation is successful, then the condition in the 5th line is satisfied. In that case the function initializes the `state` field of this element, so that it specifies a state in which both jugs contain `0` litres of water (lines 6–7). Next, the function assigns the value of the `NONE` element of the `operator_index` enumerated type to the `operator_number` field and stores the `NULL` value in the `next` field of the element (9th line). Finally, the function assigns an address of the first element to the queue pointers (10th line).

The Backtracking Algorithm — Implementation

The search() Function

```c
void search(struct queue_pointers *queue, const struct operator_structure operators[])
{
    enum operator_index operator_index;
    for(operator_index=FILL_4_LITRE; operator_index<=POUR_AND_EMPTY_3; operator_index++) {
        if(queue->tail) {
            if(operators[operator_index].is_condition_fullfiled(queue->tail->states)) {
                struct queue_node *new_state =
                create_new_state(operators[operator_index].get_next_state(queue->tail->states),
                                                                       operator_index);
                if(new_state) {
                    if(!has_already_been(*queue,new_state)) {
                        enqueue(queue,new_state);
                        if(queue->tail->states.jug_4_litre_state == 2 ||
                                        queue->tail->states.jug_3_litre_state == 2)
                            print_solution(*queue);
                        else
                            search(queue,operators);
                        remove_tail(queue);
                    } else
                        free(new_state);
                }
            }
        }
    }
}
```

The Backtracking Algorithm — Implementation

The `search()` Function

The definition of the `search()` function corresponds to the pseudocode that describes the backtracking algorithm that finds all solutions of the water jug problem and that has been presented at the beginning of the lecture. The function returns no value and takes as arguments the address of the queue pointers structure and the operators array. Please notice, that the latter argument is passed by a constant parameter. In the 3rd line of the function a local variable is declared that is used as a `for` loop counter. The loop iterates over the operators array. In each iteration of the loop the function verifies if the last element of the queue exists (5th line). If so, then it checks if the operator specified by the `operator_index` variable can be applied to the state described by this element in order to create a new state. If so, then the function calls the `create_new_state()` function to create a new element of the queue that describes this new state (lines 7–9).

The Backtracking Algorithm — Implementation

The `search()` Function

If the new element is created successfully, what is verified in the 10th line, then the function checks if the state stored in this element hasn't been found yet. To this end it calls the `has_already_been()` function and negates the returned result (11th line). If it turns up that the state has been already found then the function removes the new element from the queue (20th line) and begins a new iteration of the `for` loop. However, if the state has not been found yet then the function adds the new element to the queue (12th line) and checks if it specifies a state that satisfies the goal condition (lines 13–14). If so, then the function invokes the `print_solution()` function, to display the information about the discovered solution. Otherwise the function calls itself recursively (17th line), to check which of the operators can be applied to the state described by the newly added queue element and what new states can be created that way.

The Backtracking Algorithm — Implementation

The `search()` Function

After the function calls itself recursively or displays the details of a solution the last element of the queue is removed (18th line), so a new iteration of the `for` loop can verify if other operators can be applied to the state stored in the previous element of the queue and if other solutions of the water jug problem can be found that way.

The Backtracking Algorithm — Implementation

The `main()` Function

```c
int main(void)
{
    initialize_queue(&queue);
    search(&queue,operators);
    remove_queue(&queue);
    return 0;
}
```

The Backtracking Algorithm — Implementation

The `main()` Function

Only three of the earlier defined functions need to be invoked in the `main()` function. The first one is the `initialize_queue()` function, which adds the element specifying the initial state of jugs to the empty queue. It is called in the 3rd line. In the 4th line is invoked the `search()` function that finds all possible solutions of the water jug problem and prints their details on the screen. Finally, in the 5th line is called the `remove_queue()` function that deletes the queue left after the `search()` function exits.

Summary

The presented program finds 10 solutions of the water jug problem. Some of them are suboptimal, i.e. they have some redundant steps. Indeed, the backtracking algorithm finds all possible solutions of a problem without evaluating their quality. It is called a *brute-force* approach. To make the algorithm choose only the optimal solutions some heuristic functions should be applied that would evaluate the usefulness of each step. Initially, the backtracking algorithms were used only in the field of Artificial Intelligence, but nowadays they are applied in many other branches of Computer Science, to solve problems like: finding the extrema of multi-variable functions or performing *parsing*, i.e. the syntax analysis in the process of compiling a program.

Questions

?

The End

Thank You For Your Attention!