



الگوریتم بالا از شبه کد کتاب راسل الهام گرفته شده است:

```

function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution or failure
    solution, fvalue  $\leftarrow$  RBFS(problem, NODE(problem.INITIAL),  $\infty$ )
    return solution

function RBFS(problem, node, f limit) returns a solution or failure, and a new f-cost limit
    if problem.IS-GOAL(node.STATE) then return node
    successors  $\leftarrow$  LIST(EXPAND(node))
    if successors is empty then return failure,  $\infty$ 
    for each s in successors do // update f with value from previous search
        s.f  $\leftarrow$  max(s.PATH-COST + h(s), node.f)
    while true do
        best  $\leftarrow$  the node in successors with lowest f-value
        if best.f > f limit then return failure, best.f
        alternative  $\leftarrow$  the second-lowest f-value among successors
        result, best.f  $\leftarrow$  RBFS(problem, best, min(f limit, alternative))
        if result  $\neq$  failure then return result, best.f

```

Figure 3.22 The algorithm for recursive best-first search.

یک نمونه خروجی:

```


Run: Q1_8-puzzle x
C:\Users\Kebri\PycharmProjects\AI-HW2\venv\Sc
Initial state:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
Goal state reached:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Process finished with exit code 0

```

(۲)

الف) تابع الگوریتم minimax عادی:



```
def nodeNameOf(index):
    return chr(index + 65 - 1)

def minimax(depth, index, is_max_turn, values, path=None):
    tree_depth = int(math.log(len(values), 2))

    if path is None:
        path = []
    if depth == tree_depth:
        index = index - 2**tree_depth
        return values[index], path + [values[index]]

    best_value = None
    best_path = None

    for i in range(2):
        child_value, child_path = minimax(depth + 1, index * 2 + i, not is_max_turn, values, path +
[nodeNameOf(index)])
        if best_value is None:
            best_value = child_value
            best_path = child_path
        elif is_max_turn and child_value > best_value:
            best_value = child_value
            best_path = child_path
        elif not is_max_turn and child_value < best_value:
            best_value = child_value
            best_path = child_path

    return best_value, best_path
```

در این تابع به صورت بازگشتی فرزندان چپ و راست هر نود بررسی می شوند و از بین آنها بهترین حالت برای بازیکن max (if *is\_max\_turn* == *True*) انتخاب می شود. تابع دارای یک پارامتر path نیز هست که در آن لیست مسیری که به بهترین امتیاز ختم میشود را ذخیره میکند.

## تابع الگوریتم minimax با هرس آلفا بتا:



```
def alpha_beta(depth, index, is_max_turn, values, alpha, beta, path=None, show_pruning=False):
    tree_depth = int(math.log(len(values), 2))

    if path is None:
        path = []
    if depth == tree_depth:
        index = index - 2**tree_depth
        return values[index], path + [values[index]]

    best = MIN if is_max_turn else MAX
    best_path = []

    for i in range(2):
        value, child_path = alpha_beta(depth + 1, index * 2 + i, not is_max_turn, values, alpha, beta, path + [nodeNameOf(index)])

        if is_max_turn:
            best = max(best, value)
            alpha = max(alpha, best)
        else:
            best = min(best, value)
            beta = min(beta, best)

        if beta <= alpha:
            if (show_pruning):
                print("Pruning node", nodeNameOf(index), "with alpha =", alpha, "and beta =", beta, "at depth", depth)
            break

        if value == best:
            best_path = child_path

    return best, best_path
```

توضیحات این تابع مانند تابع قبل است. با این تفاوت که در آلفا بتا های ناخواسته، زیر درخت نا مطلوب قطع شده و دیگر بررسی نمی شود.

همچنین هنگام هرس کردن، نودی که هرس در آن رخ می دهد به همراه آلفا و بتا و عمق آن چاپ می شود.

یک توضیح کلی درباره دو تابع بالا نحوه index بندی و نام گذاری نود ها:

ایندکس root برابر ۱ در نظر گرفته شده و اگر فرض کنیم ایندکس نود  $i$  است، آنگاه ایندکس فرزند چپ برابر  $2i$  و ایندکس فرزند راست برابر  $2i + 1$  است.

طبق همین ایندکس ها طبق حروف الفبا از A تا ... نامگذاری به صورت خودکار انجام شده است. در واقع اگر یک پیمایش BFS روی درخت انجام دهیم خروجی اسم گره ها به شکل  $A, B, C, D, \dots$  خواهد بود.

## قسمت main برنامه:



```

MIN, MAX = float('-inf'), float('inf')

if __name__ == "__main__":
    is_max_turn = True
    # values = [1, 2, 3, 4, 5, 6, 7, 8] # binary tree
    values = [int(x) for x in input("Enter a binary tree (a power 2 number of nodes): ").split(",
    ")]
    print("\n===== Start of Normal Minimax =====")
    best_score, path = minimax(0, 1, is_max_turn, values)
    print("Best score:", best_score)
    print("Path to the best score:", path)

    print("\n===== Start of Minimax with Alpha Beta Pruning =====")
    best_score, path = alpha_beta(0, 1, is_max_turn, values, MIN, MAX, show_pruning=True)
    print("Best score:", best_score)
    print("Path to the best score:", path)

```

یک مثال حل شده توسط کد و مقایسه آن با حل چک شده و دستی:

```

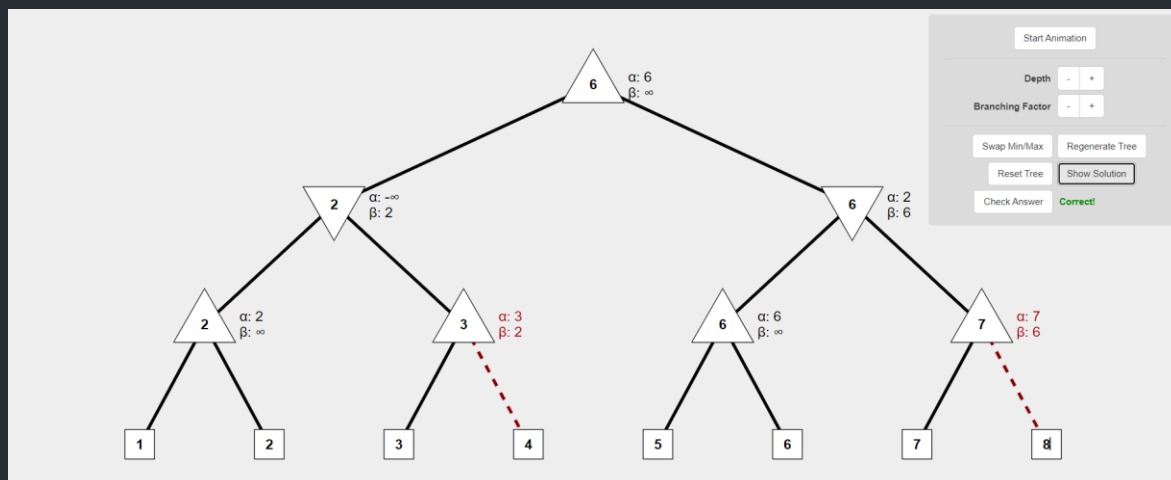
Run: Q2_A x
C:\Users\Kebri\PycharmProjects\AI-HW2\venv\Scripts\python.exe C:/Users/Kebri/PycharmProject
Enter a binary tree (a power 2 number of nodes): 1, 2, 3, 4, 5, 6, 7, 8

===== Start of Normal Minimax =====
Best score: 6
Path to the best score: ['A', 'C', 'F', 6]

===== Start of Minimax with Alpha Beta Pruning =====
Pruning node E with alpha = 3 and beta = 2 at depth 2
Pruning node G with alpha = 7 and beta = 6 at depth 2
Best score: 6
Path to the best score: ['A', 'C', 'F', 6]

Process finished with exit code 0

```





## شرح توابع اصلی:

- Fitness\_function: همان مقدار نهایی الگوریتم minimax است که در بخش قبل پیاده کردیم
- Create\_individuals: به طول لیست مان به طور رندمان هر المنت را از ۱ تا رنج داده شده مقدار دهی میکند
- Selection: به طور رندوم یک عضو از یک نسل انتخاب میشود
- Crossover: نصفی از آرایه یک پدر با نصف دیگر پدر دیگر مرج میشود
- Mutation: به صورت رندوم، یک المنت از آرایه از ۱- تا ۱ تغییر می کند

## کد بخش main:

```

if __name__ == "__main__":
    MIN, MAX = float('-inf'), float('inf')

    list_length = 8
    list_range = 8

    population_size = 100
    generations = 100
    best_solution, best_fitness = genetic_algorithm(population_size, generations)
    print("Found solution:", best_solution)
    print("Solution fitness:", best_fitness)

```

میتوان با تغییر دادن list\_length و list\_range الگوریتم را بر اساس ورودی و یا درخت دلخواه خود تنظیم نمود. دو فیلد بعدی نیز برای افزایش و یا کاهش دقت الگوریتم قابل تغییر هستند.

یک نمونه از اجرای این الگوریتم برای درختی همانند نمونه بخش الف:

```

Run: Q2_B x
C:\Users\Kebri\PycharmProjects\AI-HW2\venv\Scr
Found solution: [1, 8, 8, 1, 4, 7, 3, 6]
Solution fitness: 8

Process finished with exit code 0

```

(۳)

تابع های کمکی پیاده شده برای پیدا کردن سلول خالی (0) و تابع های کمکی برای بررسی در تناقض نبودن قرار دادن عدد n در خانه ی row و col:



```
x
def find_empty_cell(): # O(n)
    for row in range(9):
        for col in range(9):
            if grid[row][col] == 0:
                return row, col
    return None

def is_in_square(row, col, n): # O(n)
    for i in range(3):
        for j in range(3):
            if grid[row - row % 3 + i][col - col % 3 + j] == n:
                return False
    return True

def is_in_col(col, n): # O(n)
    for row in range(9):
        if grid[row][col] == n:
            return False
    return True

def is_in_row(row, n): # O(n)
    for col in range(9):
        if grid[row][col] == n:
            return False
    return True

def is_putting_this_number_valid(row, col, n): # O(n)
    if is_in_row(row, n) and is_in_col(col, n) and is_in_square(row, col, n):
        return True
    return False
```



تابع اصلی که در آن الگوریتم back-tracking برای حل سودوکو پیاده شده است:



```
def solve_sudoku():
    empty_cell = find_empty_cell()
    if empty_cell is None:
        return True
    else:
        row, col = empty_cell

    for n in range(1, 10):
        if is_putting_this_number_valid(row, col, n):
            grid[row][col] = n
            if solve_sudoku():
                return True
            grid[row][col] = 0

    return False
```

خروجی به ازای نمونه قرار داده شده در کد:

7	8	5	4	3	9	1	2	6
6	1	2	8	7	5	3	4	9
4	9	3	6	2	1	5	7	8
8	5	7	9	4	3	2	6	1
2	6	1	7	5	8	9	3	4
9	3	4	1	6	2	7	8	5
5	7	8	3	9	4	6	1	2
1	2	6	5	8	7	4	9	3
3	4	9	2	1	6	8	5	7