# Alpha-Beta Pruning

Gagarine Yaikhom

## I. Introduction

We implement the *Alpha-Beta pruning* algorithm for reducing search space in MinMax search.

⟨*Data structures*⟩≡

```
typedef struct node_s node_t;
struct node_s {
    int value;
    int num_children;
    node_t* first_child;
    node_t* last_child;
    node_t* next;
    node_t* best;
};
```

⟨*Functions*⟩≡

```
node_t *create_node()
{
    node_t* n = (node_t*)
        calloc(1, sizeof(node_t));
    if (n != NULL) {
        n->value = -1;
        n->num_children = 0;
        n->first_child = NULL;
        n->last_child = NULL;
        n->next = NULL;
        n->best = NULL;
    }
    return n;
}
```

⟨*Functions*⟩+≡

```
void append_child(node_t* t, node_t *c)
{
    if (t->first_child == NULL) {
        t->first_child = c;
        t->last_child = c;
    } else {
        t->last_child->next = c;
        t->last_child = c;
    }
    ++(t->num_children);
}
```

⟨*Macro definitions*⟩≡

```
#define SUCCESS 0
#define FAILED_TO_CREATE_NODE 1
#define INCOMPLETE_TREE_SPECIFICATION 3
#define NO_VALUE_PARSED 4
```

⟨*Functions*⟩+≡

```
int parse_value(int *value, char **s)
{
    char *p = *s;
    while (*p != '\0') {
        if (*p < '0' || *p > '9') {
            ++p;
        } else {
            break;
        }
    }
    if (*p == '\0') {
        return NO_VALUE_PARSED;
    }
    char *q = p + 1;
    while (*q != '\0') {
        if (*q >= '0' && *q <= '9') {
            ++q;
        } else {
            *q = '\0';
            break;
        }
    }

    *value = atoi(p);
    *s = q + 1;

    return SUCCESS;
}
```

⟨*Functions*⟩+≡

```
void destroy_subtree(node_t *t)
{
    if (t->num_children != 0) {
        node_t *p = t->first_child;
        while (p != NULL) {
            node_t *q = p->next;
            destroy_subtree(p);
            p = q;
        }
    }
    free(t);
}
```

⟨*Functions*⟩+≡

```
void destroy_tree(node_t *t)
{
    if (t == NULL) {
        printf("Empty tree");
    } else {
        destroy_subtree(t);
    }
}
```

⟨*Functions*⟩+≡

```
void print_indent(int depth)
{
    if (depth != 0) {
        for (int i = depth - 1; i; --i) {
            printf("    ");
        }
        printf("|___");
    }
}
```

⟨*Functions*⟩+≡

```
void print_subtree(node_t *t, int depth)
{
    if (t->num_children == 0) {
        print_indent(depth);
        printf("%d\n", t->value);
    } else {
        print_indent(depth);
        printf("[%d] %d\n",
               t->value, t->num_children);
        node_t *p = t->first_child;
        while (p != NULL) {
            print_subtree(p, depth + 1);
            p = p->next;
        }
    }
}
```

⟨*Functions*⟩+≡

```
void print_tree(node_t *t)
{
    if (t == NULL) {
        printf("Empty tree");
    } else {
        print_subtree(t, 0);
    }
}
```

⟨*Functions*⟩+≡

```
int build_subtree(
    node_t **t,
    char **s,
    int *num_inodes
)
{
    int num_children = 0;
    int r = parse_value(&num_children, s);
    if (r != SUCCESS) {
        return r;
    }
    node_t *n = create_node();
    if (n == NULL) {
        return FAILED_TO_CREATE_NODE;
    }

    if (*t == NULL) {
        *t = n;
    } else {
        append_child(*t, n);
    }
    if (num_children == 0) {
        r = parse_value(&(n->value), s);
        if (r != SUCCESS) {
            return r;
        }
    } else {
        n->value = ++(*num_inodes);
        for (int i = 0; i < num_children; ++i) {
            r = build_subtree(&n, s, num_inodes);
            if (r != SUCCESS) {
                return r;
            }
        }
    }
    return SUCCESS;
}
```

⟨*Search maximising value*⟩≡
```
value = INT_MIN;
node_t *p = t->first_child;
node_t *b = p;
while (p != NULL) {
    int v = alpha_beta(p, alpha, beta, false);
    if (v > value) {
        value = v;
        b = p;
    }
    if (value >= beta) {
        break;
    }
    p = p->next;
}
if (value > alpha) {
    t->best = b;
}
```

⟨*Search minimising value*⟩≡
```
value = INT_MAX;
node_t *p = t->first_child;
node_t *b = p;
while (p != NULL) {
    int v = alpha_beta(p, alpha, beta, true);
    if (v < value) {
        value = v;
        b = p;
    }
    if (value <= alpha) {
        break;
    }
    p = p->next;
}
if (value < beta) {
    t->best = b;
}
```

⟨*Functions*⟩+≡
```
int alpha_beta(node_t *t, int alpha, int beta, bool maximising)
{
    if (t->num_children == 0) {
        return t->value;
    }
    int value = 0;
    if (maximising) {
        ⟨Search maximising value⟩;
    } else {
        ⟨Search minimising value⟩;
    }
    return value;
}
```

⟨*Functions*⟩+≡
```
void print_best_path(node_t *t)
{
    if (t->best == NULL) {
        printf("%d\n", t->value);
    } else {
        printf("[%d]->", t->value);
        print_best_path(t->best);
    }
}
```

⟨*Test functions*⟩≡
```
int test_parse_value(char *s)
{
    char *p = s;
    int value = -1;
    while (parse_value(&value, &p) == SUCCESS) {
        printf("%d\n", value);
    }

    return SUCCESS;
}
```

⟨*Test functions*⟩+≡
```
int test_tree_builder(char *s)
{
    int num_inodes = 0;
    node_t *tree = NULL;
    int r = build_subtree(&tree, &s, &num_inodes);
    if (r != SUCCESS) {
        return r;
    }
    print_tree(tree);
    destroy_tree(tree);
    return SUCCESS;
}
```

⟨*Test functions*⟩+≡

```
int test_alpha_beta(char *s)
{
    int num_inodes = 0;
    node_t *tree = NULL;
    int r = build_subtree(&tree, &s, &num_inodes);
    if (r != SUCCESS) {
        return r;
    }
    print_tree(tree);

    int alpha = INT_MIN;
    int beta = INT_MAX;
    int value = alpha_beta(tree, alpha, beta, true);
    printf("Value: %d\nBest path: ", value);
    print_best_path(tree);

    destroy_tree(tree);
    return SUCCESS;
}
```

## II. Main program

Main program for testing the functions.

⟨*Main*⟩≡

```
int main(int argc, char **argv) {
    /* test_parse_value(argv[1]); */
    /* test_tree_builder(argv[1]); */
    test_alpha_beta(argv[1]);
    return 0;
}
```

⟨*Standard libraries*⟩≡

```
#include <limits.h>
#include <malloc.h>
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
```

⟨*algorithm.c*⟩≡

⟨*Standard libraries*⟩
⟨*Macro definitions*⟩
⟨*Data structures*⟩
⟨*Functions*⟩
⟨*Test functions*⟩
⟨*Main*⟩

## III. Running the program

Run the program as follows:

```
./alpha_beta "search space"
```

Here, the search space is specified left-to-right depth first search approach where we specify the number of children for each node to be inserted, and continue inserting subtrees until we insert a leaf, which is preceded by zero number of children. The following are few example:

```
$ ./algorithm "3 2 2 0 1 0 2 3 0 3 0 4 0 5 3 2 0 6 0 7 1 0 8 2 0 9 0 \
10 4 0 11 2 0 12 2 0 13 0 14 1 0 15 1 1 0 16"
[1] 3
|___[2] 2
    |___[3] 2
        |___1
        |___2
    |___[4] 3
        |___3
        |___4
        |___5
|___[5] 3
    |___[6] 2
        |___6
        |___7
    |___[7] 1
        |___8
    |___[8] 2
        |___9
        |___10
|___[9] 4
    |___11
    |___[10] 2
        |___12
        |___[11] 2
            |___13
            |___14
    |___[12] 1
        |___15
    |___[13] 1
        |___[14] 1
            |___16
Value: 11
Best path: [1]->[9]->11
```