Seven algorithms to solve 8 puzzle

by Liu Kuan 12086075

Features

- `FunctionDictionary` adopted `std::unorded_map` i.e. a hash table as it's container. Due to no mutation at run time, it guarantees the best lookup performance.
- By using `std::unordered_set`, all alorithms that have visted list inside are able to operate on it at the smallest time complexity.
- A custom defined priority queue was implemented for "uniform cost search" and "A* with strict expanded list" to precisely implement them as required.
- Following Object Orientation and Functional programming paradigms, this code provides better readability, modularity, abstraction and maintainability.
- Following TTD(Test-driven development), the unit tests covering all code guarantee everything has been done as expected.
- Pure C++ 11.

Pseudocode

```
//
//@filename = "node.hpp"
//
>>>>>>> 44346eaaf2125ecc45f5f4dd47023ecc81755087
let Node be:
    state
    path
as struct


//
//@filename = "default_cost_func.hpp"
//
let DefaultCostFunc (node) be:
    return size(path(node))
as functor


//
//@filename = "priority_queue.hpp"
//
[Using pseudocodes from "Chapter 6, Introduction to Algorithms 3rd edition" aka
C.L.R.S.]


//
//@filename = "time_record.hpp"
//note = "an RAII style timer"
//
let TimeRecord be:
```

```
    let constructor(reference) be:
        start timer

    let destructor() be:
        stop timer and write time duration to outside by reference
as class


//
//@filename = "heuristic_func.hpp"
//
let ManhattanDistance (curr, goal) be:
    ret = 0
    for i = 0 to length(goal) - 1
        if  '0' != curr[i]
            digit = curr[i] - '0'
            ret = ret + abs(i / 3 - digit / 3) + abs(i % 3 - digit % 3)
    return ret
as functor

let MisplacedTiles (curr, goal) be :
    count = 0
    for i = 0 to length(goal) - 1
        if  curr[i] != goal[i]
            increment(count)
    return count
as functor


//
//@file_name = function_dictionary.hpp
//@note = this class implmented a function dictionary mapping each position of `0`
to its possible children state.
//
let FunctionDictionary be:
    let FunctionDictionary() be:
        fill_dictionary()
    as constructor

    let fill_dictionary() be:
        let u(position) = position - 3 as lambda
        let d(position) = position + 3 as lambda
        let l(position) = position - 1 as lambda
        let r(position) = position + 1 as lambda

        let make_child(parent, move_lambda, direction) be:
            pos = state(parent).find('0')
            stt = state(parent)
            swap(stt[pos], stt[move_lambda(pos)])
            return Node(stt, path(parent) + direction)
        as lambda

        let up(parent) = make_child(parent, u, 'U') as lambda
        let dw(parent) = make_child(parent, d, 'D') as lambda
        let lt(parent) = make_child(parent, l, 'L') as lambda
        let rt(parent) = make_child(parent, r, 'R') as lambda
```

```
        //fill possible lambda into dictionary
        this[0] = LambdaList{ dw, rt }
        this[2] = LambdaList{ dw, lt }
        this[6] = LambdaList{ up, rt }
        this[8] = LambdaList{ up, lt }
        this[1] = LambdaList{ dw, lt, rt }
        this[3] = LambdaList{ up, dw, rt }
        this[5] = LambdaList{ up, dw, lt };
        this[7] = LambdaList{ up, lt, rt };
        this[4] = LambdaList{ up, dw, lt, rt };
    as method
as class


//
//@filename = progressive_deepening_search_with_visited_list.hpp`
//
let PDSWithVList be:
    let PDSWithVList(source, goal) be:
        record time
        search(source, goal)
    as constructor

    let search(source, goal) be:
        max_depth = 0
        while true
            reset q and visited_list
            q.push(Node(source))
            while q is not empty
                curr = pop(q)
                visited_list.insert(state(curr))
                if goal == state(curr)
                    final_path = path(curr), return
                if length(path(curr)) < max_depth
                    let func_list point to: function_dictionary[find position of
'0' in state(curr)]) as referrence
                    for each make_child as lambda in funct_list
                        child = make_child(curr)
                        if visited_list doesn't contain state(child)
                            q.push(child)
                max_q_length = max(max_q_length, size(q))
    as method
as class


//
//@filename = "best_first_search_with_visited_list.hpp"
//
let BestFSWithVList be:
    //this functor is going to be passed to priority queue for comparison
    let Greater(lhs, rhs) be:
        let h be an object as HeuristicFunc
        return h(lhs to goal) > h(rhs to goal)
    as functor

    let BestFSWithVList(source, goal) be:
        record time
```

```
        search(source, goal)
    as constructor

    let search(source, goal) be:
        q.push(Node(source))
        while q is not empty
            curr = q.pop()
            visited_list.insert(state(curr))
            if goal == state(curr)
                final_path = path(curr), return
            let func_list point to: function_dictionary[find position of '0' in
state(curr)]) as referrence
            for each make_child as lambda in func_list
                child = make_child(curr)
                if visited_list doesn't contain state(child)
                    q.push(child)
            max_q_length = max(max_q_length, size(q))
    as method
as class


//
//@filename = "UniformCostSearch.hpp"
//
let UniformCostSearch be:
    let Shorter(lhs, rhs) be:
        return length(path(lhs)) > length(path(rhs))
    as functor

    let UniformCostSearch(source, goal) be:
        record time
        search(source, goal)
    as constructor

    let search(source, goal) be:
        q.push(source)
        while q is not empty
            if goal == state(curr)
                final_path = path(curr), return
            if expanded_list doesn't contain state(curr)
                expanded_list.insert(state(curr))
                let func_list point to: function_dictionary[find position of '0' in
state(curr)]) as referrence
                for each make_child as lambda in func_list
                    child = make_child(curr)
                    if expanded_list doesn't contain state(child)
                        find it as iterator in q, such that: state(child) ==
state(node)
                        if it doesn't exist
                            q.push(child)
                        else if it has lower cost than child has
                            swap(the node it pointing to, child)
            max_q_length = max(max_q_length, size(q))
    as method
as class

//
```

```
//@filename = "UniformCostSearch.hpp"
//
let AStar be:
    let Greater(lhs, rhs) be:
        let h be an object of HeuristicFunc
        let c be an object of CostFunc
        return h(state(lhs), goal) + c(lhs) > h(state(lhs), goal) + c(rhs)
    as functor

    let AStar(source,goal) be:
        record time
        search(source, goal)
    as constructor

    let search(source, goal) be:
        q.push(Node(source))
        while q is not empty
            curr = pop(q)
            if state(curr) == goal
                final_path = path(curr),return
            let func_list point to: function_dictionary[find position of '0' in
state(curr)]) as referrence
            for each make_child as lambda in func_list
                q.push(make_child(curr))
            max_q_length = max(max_q_length, size(q))
    as method
as class

//
//@filename = "a_star_with_strict_expanded_list.hpp"
//
let AStarSEL be:
   let Less(lhs, rhs) be:
        let h be an object of HeuristicFunc
        let c be an object of CostFunc
        return h(state(lhs), goal) + c(lhs) < h(state(lhs), goal) + c(rhs)
    as functor

    let AStarSEL(source,goal) be:
        record time
        search(source, goal)
    as constructor

    let search(source, goal) be:
        let less = Less(goal)
        q.push(Node(source))
        while q is not empty
            curr = pop(q)
            if state(curr) == goal
                final_path = path(curr), return
            if expanded_list doesn't contain state(curr)
                expanded_list.insert(state(curr))
                let func_list point to function_dictionary[find position of '0' in
state(curr)]) as referrence
                for each make_child as lambda in func_list
                    child = make_child(curr)
                    if expanded_list doesn't contain state(child)
```

```
                        find iter as iterator in q, such that: state(child) ==
state(dereference(iter))
                        if iter doesn't exist
                            q.push(child)
                        else if less(child, dereference(iter))
                            swap(child, dereference(iter))
            max_q_length = max(max_q_length, size(q))
    as method
as class
```