VE281 Project Three Report

Liu Yihao 515370910207

1 Introduction

In order to study the performances of these three priority queues, I generated random inputs with different grid sizes and compared the running speed of them (including std::priority_queue in STL). Since it's a waste of time to wrote a comparison script written in C++, I chose node-gyp to build the sorting algorithm into a C++ addon of node, and then wrote some Javascript code to benchmark them. Small size of arrays were run for several times so that the result can be more accurate.

2 Comparison of algorithms

The limitation of runtime was set to 1s for all algorithms, so some meaningless and slow running were dropped. Then I used MATLAB to plot a graph.

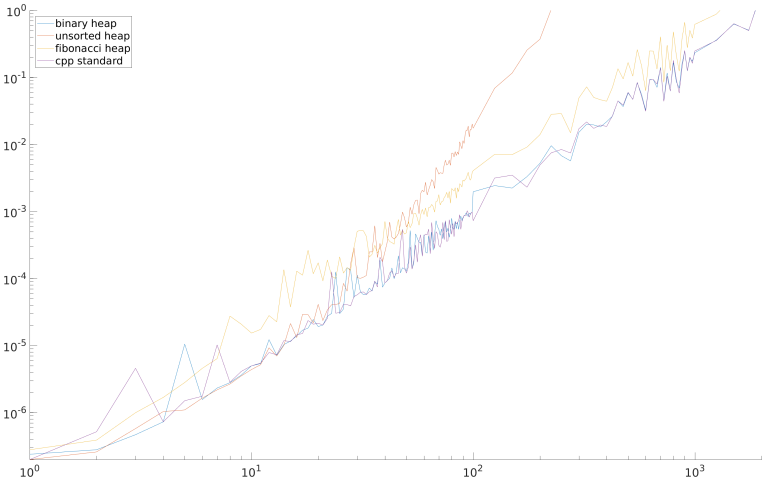

Figure 1: All cases

From Figure 1, we can find that binary_heap, fib_heap and std::priority_queue have the similar running speed. The result satisfy the theory that they have the time complexity of $O(\log n)$ in enqueing, finding minimum and dequeing minimum. However, unsorted_heap is slower than those three queues, because it have the time complexity of $O(n)$ in finding minimum and dequeing minimum.

3 Appendix

3.1 The project files

3.1.1 binary_heap.h

```
#ifndef BINARY_HEAP_H
#define BINARY_HEAP_H

#include <algorithm>
#include "priority_queue.h"

// OVERVIEW: A specialized version of the 'heap' ADT implemented as a binary
//           heap.
template<typename TYPE, typename COMP = std::less<TYPE> >
class binary_heap : public priority_queue<TYPE, COMP> {
public:
    typedef unsigned size_type;

    // EFFECTS: Construct an empty heap with an optional comparison functor.
    //          See test_heap.cpp for more details on functor.
    // MODIFIES: this
    // RUNTIME: O(1)
    binary_heap(COMP comp = COMP());

    // EFFECTS: Add a new element to the heap.
    // MODIFIES: this
    // RUNTIME: O(log(n))
    virtual void enqueue(const TYPE &val);

    // EFFECTS: Remove and return the smallest element from the heap.
    // REQUIRES: The heap is not empty.
    // MODIFIES: this
    // RUNTIME: O(log(n))
    virtual TYPE dequeue_min();

    // EFFECTS: Return the smallest element of the heap.
    // REQUIRES: The heap is not empty.
    // RUNTIME: O(1)
    virtual const TYPE &get_min() const;

    // EFFECTS: Get the number of elements in the heap.
    // RUNTIME: O(1)
    virtual size_type size() const;

    // EFFECTS: Return true if the heap is empty.
    // RUNTIME: O(1)
    virtual bool empty() const;

private:
```

```
45      // Note: This vector *must* be used in your heap implementation.
46      std::vector<TYPE> data;
47      // Note: compare is a functor object
48      COMP compare;
49
50  private:
51      // Add any additional member functions or data you require here.
52  };
53
54  template<typename TYPE, typename COMP>
55  binary_heap<TYPE, COMP>::binary_heap(COMP comp) {
56      compare = comp;
57      // Fill in the remaining lines if you need.
58      data.push_back(std::move(TYPE()));
59  }
60
61  template<typename TYPE, typename COMP>
62  void binary_heap<TYPE, COMP>::enqueue(const TYPE &val) {
63      // Fill in the body.
64      data.push_back(val);
65      auto id = this->size();
66      while (id > 1 && compare(data[id], data[id / 2])) {
67          std::swap(data[id], data[id / 2]);
68          id /= 2;
69      }
70  }
71
72  template<typename TYPE, typename COMP>
73  TYPE binary_heap<TYPE, COMP>::dequeue_min() {
74      // Fill in the body.
75      if (this->empty()) return data[0];
76      auto val = data[1];
77      data[1] = data.back();
78      data.pop_back();
79      auto n = data.size();
80      for (size_type id = 1, j = id * 2; j <= n; j = 2 * id) {
81          if (j < n && compare(data[j + 1], data[j]))j++;
82          if (!compare(data[j], data[id])) break;
83          std::swap(data[id], data[j]);
84          id = j;
85      }
86      return val;
87  }
88
89  template<typename TYPE, typename COMP>
90  const TYPE &binary_heap<TYPE, COMP>::get_min() const {
91      // Fill in the body.
92      if (this->empty()) return data[0];
93      return data[1];
```

```
}

template<typename TYPE, typename COMP>
bool binary_heap<TYPE, COMP>::empty() const {
    // Fill in the body.
    return this->size() == 0;
}

template<typename TYPE, typename COMP>
unsigned binary_heap<TYPE, COMP>::size() const {
    // Fill in the body.
    return data.size() - 1;
}

#endif //BINARY_HEAP_H
```

3.1.2 unsorted_heap.h

```
#ifndef UNSORTED_HEAP_H
#define UNSORTED_HEAP_H

#include <algorithm>
#include "priority_queue.h"

// OVERVIEW: A specialized version of the 'heap' ADT that is implemented with
//           an underlying unordered array-based container. Every time a min
//           is required, a linear search is performed.
template<typename TYPE, typename COMP = std::less<TYPE> >
class unsorted_heap : public priority_queue<TYPE, COMP>
{
public:
    typedef unsigned size_type;

    // EFFECTS: Construct an empty heap with an optional comparison functor.
    //          See test_heap.cpp for more details on functor.
    // MODIFIES: this
    // RUNTIME: O(1)
    unsorted_heap(COMP comp = COMP());

    // EFFECTS: Add a new element to the heap.
    // MODIFIES: this
    // RUNTIME: O(1)
    virtual void enqueue(const TYPE &val);

    // EFFECTS: Remove and return the smallest element from the heap.
    // REQUIRES: The heap is not empty.
    // MODIFIES: this
    // RUNTIME: O(n)
    virtual TYPE dequeue_min();

```

```
33      // EFFECTS: Return the smallest element of the heap.
34      // REQUIRES: The heap is not empty.
35      // RUNTIME: O(n)
36      virtual const TYPE &get_min() const;
37
38      // EFFECTS: Get the number of elements in the heap.
39      // RUNTIME: O(1)
40      virtual size_type size() const;
41
42      // EFFECTS: Return true if the heap is empty.
43      // RUNTIME: O(1)
44      virtual bool empty() const;
45
46  private:
47      // Note: This vector *must* be used in your heap implementation.
48      std::vector<TYPE> data;
49      // Note: compare is a functor object
50      COMP compare;
51  private:
52      // Add any additional member functions or data you require here.
53      TYPE error;
54  };
55
56  template<typename TYPE, typename COMP>
57  unsorted_heap<TYPE, COMP>::unsorted_heap(COMP comp)
58  {
59      compare = comp;
60      // Fill in the remaining lines if you need.
61  }
62
63  template<typename TYPE, typename COMP>
64  void unsorted_heap<TYPE, COMP>::enqueue(const TYPE &val)
65  {
66      // Fill in the body.
67      data.push_back(val);
68  }
69
70  template<typename TYPE, typename COMP>
71  TYPE unsorted_heap<TYPE, COMP>::dequeue_min()
72  {
73      // Fill in the body.
74      if (this->empty()) return this->error;
75      auto it = std::min_element(data.begin(), data.end(), compare);
76      std::swap(*it, data.back());
77      auto min = std::move(data.back());
78      data.pop_back();
79      return min;
80  }
81
```

```
82  template<typename TYPE, typename COMP>
83  const TYPE &unsorted_heap<TYPE, COMP>::get_min() const
84  {
85      // Fill in the body.
86      if (this->empty()) return this->error;
87      auto it = std::min_element(data.begin(), data.end(), compare);
88      return *it;
89  }
90
91  template<typename TYPE, typename COMP>
92  bool unsorted_heap<TYPE, COMP>::empty() const
93  {
94      // Fill in the body.
95      return data.empty();
96  }
97
98  template<typename TYPE, typename COMP>
99  unsigned unsorted_heap<TYPE, COMP>::size() const
100 {
101     // Fill in the body.
102     return data.size();
103 }
104
105 #endif //UNSORTED_HEAP_H
```

3.1.3 fib_heap.h

```
1   #ifndef FIB_HEAP_H
2   #define FIB_HEAP_H
3
4   #include <algorithm>
5   #include <cmath>
6   #include <list>
7   #include "priority_queue.h"
8
9   // OVERVIEW: A specialized version of the 'heap' ADT implemented as a
10  //           Fibonacci heap.
11  template<typename TYPE, typename COMP = std::less<TYPE> >
12  class fib_heap : public priority_queue<TYPE, COMP> {
13  public:
14      typedef unsigned size_type;
15
16      // EFFECTS: Construct an empty heap with an optional comparison functor.
17      //          See test_heap.cpp for more details on functor.
18      // MODIFIES: this
19      // RUNTIME: O(1)
20      fib_heap(COMP comp = COMP());
21
22      // EFFECTS: Add a new element to the heap.
23      // MODIFIES: this
```

```
24      // RUNTIME: O(1)
25      virtual void enqueue(const TYPE &val);
26
27      // EFFECTS: Remove and return the smallest element from the heap.
28      // REQUIRES: The heap is not empty.
29      // MODIFIES: this
30      // RUNTIME: Amortized O(log(n))
31      virtual TYPE dequeue_min();
32
33      // EFFECTS: Return the smallest element of the heap.
34      // REQUIRES: The heap is not empty.
35      // RUNTIME: O(1)
36      virtual const TYPE &get_min() const;
37
38      // EFFECTS: Get the number of elements in the heap.
39      // RUNTIME: O(1)
40      virtual size_type size() const;
41
42      // EFFECTS: Return true if the heap is empty.
43      // RUNTIME: O(1)
44      virtual bool empty() const;
45
46  private:
47      // Note: compare is a functor object
48      COMP compare;
49
50  private:
51      // Add any additional member functions or data you require here.
52      // You may want to define a strcut/class to represent nodes in the heap and a
53      // pointer to the min node in the heap.
54      struct node {
55          TYPE val;
56          unsigned depth = 0;
57          std::list<node> child;
58
59          explicit node(const TYPE &val) {
60              this->val = val;
61          }
62      };
63
64      typename std::list<node> root;
65      typename std::list<node>::iterator min;
66      unsigned num_elements = 0;
67      const TYPE &default_element = TYPE();
68  };
69
70  // Add the definitions of the member functions here. Please refer to
71  // binary_heap.h for the syntax.
72
```

```
73  template<typename TYPE, typename COMP>
74  fib_heap<TYPE, COMP>::fib_heap(COMP comp) {
75      compare = comp;
76      min = root.begin();
77  }
78
79  template<typename TYPE, typename COMP>
80  void fib_heap<TYPE, COMP>::enqueue(const TYPE &val) {
81      if (root.empty()) {
82          root.push_front(std::move(node(val)));
83          min = root.begin();
84      } else if (compare(val, min->val)) {
85          min = root.insert(min, std::move(node(val)));
86      } else {
87          root.insert(min, std::move(node(val)));
88      }
89      num_elements++;
90  }
91
92  template<typename TYPE, typename COMP>
93  TYPE fib_heap<TYPE, COMP>::dequeue_min() {
94      if (this->empty()) return default_element;
95      num_elements--;
96      auto temp = min->val;
97
98      if (num_elements == 0) {
99          root.clear();
100         return temp;
101     }
102
103     root.splice(min, min->child);
104     min = root.erase(min);
105     if (min == root.end()) {
106         min = root.begin();
107     }
108
109     auto it = min;
110     auto size = (unsigned) (log(num_elements) / log(1.618)) + 1;
111     bool isset[size] = {0};
112     typename std::list<node>::iterator arr[size];
113
114     auto root_size = root.size();
115     for (auto i = 0; i < root_size; i++) {
116         if (it == root.end()) it = root.begin();
117         bool flag = true;
118         auto now = it;
119         unsigned depth = now->depth;
120         while (isset[depth]) {
121             if (compare(now->val, arr[depth]->val)) {
```

```
122                 now->child.push_front(std::move(*arr[depth]));
123                 root.erase(arr[depth]);
124             } else {
125                 arr[depth]->child.push_front(std::move(*now));
126                 if (now == it) {
127                     it = root.erase(now);
128                     flag = false;
129                 } else {
130                     root.erase(now);
131                 }
132                 now = arr[depth];
133             }
134             isset[depth] = false;
135             depth++;
136             now->depth++;
137         }
138         isset[depth] = true;
139         if (now == root.end()) now = root.begin();
140         arr[depth] = now;
141         if (flag) ++it;
142     }
143     min = std::min_element(root.begin(), root.end(), [this](const node &a, const
      ↪  node &b) {
144         return compare(a.val, b.val);
145     });
146     return temp;
147 }
148
149 template<typename TYPE, typename COMP>
150 const TYPE &fib_heap<TYPE, COMP>::get_min() const {
151     // Fill in the body.
152     if (this->empty()) return default_element;
153     return min->val;
154 }
155
156 template<typename TYPE, typename COMP>
157 bool fib_heap<TYPE, COMP>::empty() const {
158     // Fill in the body.
159     return this->size() == 0;
160 }
161
162 template<typename TYPE, typename COMP>
163 unsigned fib_heap<TYPE, COMP>::size() const {
164     // Fill in the body.
165     return this->num_elements;
166 }
167
168 #endif //FIB_HEAP_H
```

3.1.4 main.cpp

```cpp
//
// Created by liu on 2017/9/15.
//

#include <iostream>
#include <fstream>
#include <string>
#include <getopt.h>

#include "binary_heap.h"
#include "unsorted_heap.h"
#include "fib_heap.h"

using namespace std;

class Point {
public:
    size_t x, y;
    size_t weight, cost = 0;
    bool reached = false;
    Point *predecessor = NULL;

    struct ptr_compare_t {
        bool operator()(const Point *a, const Point *b) const {
            if (a->cost != b->cost) return a->cost < b->cost;
            if (a->y != b->y) return a->y < b->y;
            return a->x < b->x;
        }
    };

    friend ostream &operator<<(ostream &out, const Point &p) {
        out << "(" << p.y << ", " << p.x << ")";
        return out;
    }

    Point() = default;

    ~Point() = default;
};

void print_path(ostream &out, const Point &p) {
    if (p.predecessor) print_path(out, *(p.predecessor));
    out << p << endl;
}

int main(int argc, char *argv[]) {
    bool verbose = false;
    string impl;
```

```
    while (true) {
        const option long_options[] = {
                {"verbose",        no_argument,       NULL, 'v'},
                {"implementation", required_argument, NULL, 'i'},
                {0, 0, 0,                                   0}
        };
        int c = getopt_long(argc, argv, "vi:", long_options, NULL);
        if (c == -1)break;
        if (c == 'v') {
            verbose = true;
        } else if (c == 'i') {
            impl = optarg;
        }
    }

    priority_queue<Point *, Point::ptr_compare_t> *queue;
    if (impl == "BINARY") queue = new binary_heap<Point *,
     ↪  Point::ptr_compare_t>();
    else if (impl == "UNSORTED") queue = new unsorted_heap<Point *,
     ↪  Point::ptr_compare_t>();
    else if (impl == "FIBONACCI") queue = new fib_heap<Point *,
     ↪  Point::ptr_compare_t>();
    else return 0;

    if (argc < optind) return 0;

    ifstream fin;
    ofstream fout;

    if (argc >= optind + 4) {
        if (string(argv[optind + 2]) == "<") {
            fin.open(argv[optind + 3]);
            cin.rdbuf(fin.rdbuf());
        } else if (string(argv[optind + 2]) == ">") {
            fout.open(argv[optind + 3]);
            cout.rdbuf(fout.rdbuf());
        }
    }
    if (argc >= optind + 2) {
        if (string(argv[optind]) == "<") {
            fin.open(argv[optind + 1]);
            cin.rdbuf(fin.rdbuf());
        } else if (string(argv[optind]) == ">") {
            fout.open(argv[optind + 1]);
            cout.rdbuf(fout.rdbuf());
        }
    }

    size_t m, n, x1, x2, y1, y2;
```

```
95      cin >> n >> m >> y1 >> x1 >> y2 >> x2;
96
97      Point **grid = new Point *[m];
98      for (size_t i = 0; i < m; i++) {
99          grid[i] = new Point[n];
100         for (size_t j = 0; j < n; j++) {
101             auto p = &grid[i][j];
102             cin >> p->weight;
103             p->cost = p->weight;
104             p->x = i;
105             p->y = j;
106         }
107     }
108
109     auto start = &grid[x1][y1];
110     auto end = &grid[x2][y2];
111     start->reached = true;
112     queue->enqueue(start);
113
114     size_t step = 0;
115     const int DIR_X[4] = {0, 1, 0, -1};
116     const int DIR_Y[4] = {1, 0, -1, 0};
117
118     while (!queue->empty()) {
119         auto C = queue->dequeue_min();
120         if (verbose) {
121             cout << "Step " << step << endl;
122             cout << "Choose cell " << *C << " with accumulated length " <<
                ↪  C->cost << "." << endl;
123         }
124         step++;
125         for (int i = 0; i < 4; i++) {
126             auto x = C->x + DIR_X[i];
127             auto y = C->y + DIR_Y[i];
128             if (x < 0 || x >= m || y < 0 || y >= n) continue;
129             auto N = &grid[x][y];
130             if (N->reached) continue;
131             N->cost += C->cost;
132             N->reached = true;
133             N->predecessor = &grid[C->x][C->y];
134             if (N->x == x2 && N->y == y2) {
135                 if (verbose) {
136                     cout << "Cell " << *N << " with accumulated length " <<
                        ↪  N->cost << " is the ending point."
137                          << endl;
138                 }
139                 cout << "The shortest path from " << *start << " to " << *end <<
                    ↪  " is " << N->cost << "." << endl;
140                 cout << "Path:" << endl;
```

```
                print_path(cout, *N);
                for (size_t i = 0; i < m; i++) {
                    delete[] grid[i];
                }
                delete[] grid;
                delete queue;
                fin.close();
                fout.close();
                return 0;
            } else {
                queue->enqueue(N);
                if (verbose) {
                    cout << "Cell " << *N << " with accumulated length " <<
                     ↪  N->cost << " is added into the queue."
                         << endl;
                }
            }
        }
    }
    for (size_t i = 0; i < m; i++) {
        delete[] grid[i];
    }
    delete[] grid;
    delete queue;
    fin.close();
    fout.close();
    return 0;
}
```

3.1.5 Makefile

```
all: main.cpp binary_heap.h unsorted_heap.h fib_heap.h priority_queue.h
    g++ -O3 -std=c++11 -o main main.cpp binary_heap.h unsorted_heap.h fib_heap.h
     ↪  priority_queue.h
clean:
    rm ./main
```

3.2 The benchmark program

3.2.1 README.md

```
# Benchmark of priority queues

## Introduction

The benchmark is under node, with node-gyp to build the cpp addon,
which receives test-cases and return each sorting algorithm's running time.

## Configuration

```

```
If you are testing your own cpp source, you may need to edit `queue_wrapper.h`
↪  and `binding.gyp`.

Make sure to have `node` installed, and then run

```
npm install -g node-gyp
npm install
CXXFLAGS='-std=c++14' node-gyp configure build
```

Benchmarking

If no error occurs in configuration, run this

```
node benchmark.js
```

Then you can use the MATLAB script `benchmark.m` to plot figures.
```

3.2.2 queue_wrapper.h

```
//
// Created by liu on 17-9-3.
//

#ifndef P3_BENCHMARK_WRAPPER_H
#define P3_BENCHMARK_WRAPPER_H

#include <algorithm>
#include <memory>
#include <vector>
#include <queue>
#include "../answer/priority_queue.h"
#include "../answer/binary_heap.h"
#include "../answer/unsorted_heap.h"
#include "../answer/fib_heap.h"

template<typename TYPE, typename COMP = std::less<TYPE> >
class stl_heap : public priority_queue<TYPE, COMP> {
private:
    struct REVERSE_COMP {
        COMP compare;
        bool operator()(const TYPE &a, const TYPE &b) {
            return compare(b, a);
        }
    };

    std::priority_queue<TYPE, std::vector<TYPE>, REVERSE_COMP> queue;
```

```
28      COMP compare;
29  public:
30      typedef unsigned size_type;
31      explicit stl_heap(COMP comp = COMP()) {
32          queue = std::priority_queue<TYPE, std::vector<TYPE>,
            ↪ REVERSE_COMP>(REVERSE_COMP{comp});
33      }
34      virtual void enqueue(const TYPE &val) { queue.push(val); }
35      virtual TYPE dequeue_min() {
36          auto val = queue.top();
37          queue.pop();
38          return val;
39      }
40      virtual const TYPE &get_min() const { return queue.top(); }
41      virtual size_type size() const { return queue.size(); }
42      virtual bool empty() const { return queue.empty(); }
43  };
44
45  class Point {
46  public:
47      typedef std::unique_ptr<Point> Ptr;
48
49      size_t x = 0, y = 0;
50      size_t weight = 0, cost = 0;
51      bool reached = false;
52      Point *predecessor = nullptr;
53
54      struct ptr_compare_t {
55          bool operator()(const Point *a, const Point *b) const {
56              if (a->cost != b->cost) return a->cost < b->cost;
57              if (a->y != b->y) return a->y < b->y;
58              return a->x < b->x;
59          }
60      };
61
62      Point() = default;
63      ~Point() = default;
64  };
65
66  long path_test(int32_t type, int32_t n, int32_t m, const int32_t *data) {
67      std::unique_ptr<priority_queue<Point *, Point::ptr_compare_t> > queue;
68      switch (type) {
69      case 0:
70          queue = std::make_unique<binary_heap<Point *, Point::ptr_compare_t> >();
71          break;
72      case 1:
73          queue = std::make_unique<unsorted_heap<Point *, Point::ptr_compare_t>
            ↪ >();
74          break;
```

```
    case 2:
        queue = std::make_unique<fib_heap<Point *, Point::ptr_compare_t> >();
        break;
    case 3:
        queue = std::make_unique<stl_heap<Point *, Point::ptr_compare_t> >();
        break;
    default:
        return 0;
    }

    std::vector<std::vector<Point::Ptr> > grid(m);
    int k = 0;
    for (size_t i = 0; i < m; i++) {
        for (size_t j = 0; j < n; j++) {
            auto p = std::make_unique<Point>();
            p->x = i;
            p->y = j;
            p->cost = p->weight = (size_t) data[k++];
            grid[i].push_back(std::move(p));
        }
    }

    const size_t x1 = 0, y1 = 0;
    const size_t x2 = m - 1, y2 = n - 1;

    const int DIR_X[4] = {0, 1, 0, -1};
    const int DIR_Y[4] = {1, 0, -1, 0};

    auto clock1 = clock();
    auto start = grid[x1][y1].get();
    start->reached = true;
    queue->enqueue(start);

    while (!queue->empty()) {
        auto C = queue->dequeue_min();
        for (int i = 0; i < 4; i++) {
            auto x = (int) C->x + DIR_X[i];
            auto y = (int) C->y + DIR_Y[i];
            if (x < 0 || x >= m || y < 0 || y >= n) continue;
            auto N = grid[x][y].get();
            if (N->reached) continue;
            N->cost += C->cost;
            N->reached = true;
            N->predecessor = grid[C->x][C->y].get();
            if (N->x == x2 && N->y == y2) {
                return clock() - clock1;
            } else {
                queue->enqueue(N);
            }
```

```
        }
    }
    return clock() - clock1;
}

#endif //P3_BENCHMARK_WRAPPER_H
```

3.2.3 queue_wrapper.cpp

```
#include <node.h>
#include <node_buffer.h>
#include <sstream>
#include "queue_wrapper.h"

using namespace v8;
using namespace std;
using namespace node;

void Generate(const FunctionCallbackInfo<Value> &args) {
    Isolate *isolate = args.GetIsolate();

    if (args.Length() < 2) {
        isolate->ThrowException(Exception::TypeError(
                String::NewFromUtf8(isolate, "Wrong number of arguments")));
        return;
    }

    if (!args[0]->IsString() || !args[1]->IsInt32()) {
        isolate->ThrowException(Exception::TypeError(
                String::NewFromUtf8(isolate, "Wrong arguments")));
        return;
    }

    auto arg0 = Local<String>::Cast(args[0]);
    auto arg1 = (size_t) args[1]->IntegerValue();

    auto str = new char[arg0->Length() + 1];
    arg0->WriteUtf8(str);
    hash<string> str_hash;
    auto seed = str_hash(str);
    srand48(seed);
    delete[] str;

    auto buf = Buffer::New(isolate, arg1 * 4);
    auto localBuf = buf.ToLocalChecked();
    auto data = (int32_t *) Buffer::Data(localBuf);


    for (uint32_t i = 0; i < arg1; i++) {
        data[i] = (int32_t) mrand48();
```

```
42      }
43      args.GetReturnValue().Set(localBuf);
44  }
45
46
47  void Queue(const FunctionCallbackInfo<Value> &args) {
48      Isolate *isolate = args.GetIsolate();
49
50      if (args.Length() < 2) {
51          // Throw an Error that is passed back to JavaScript
52          isolate->ThrowException(Exception::TypeError(
53                  String::NewFromUtf8(isolate, "Wrong number of arguments")));
54          return;
55      }
56
57      if (!args[1]->IsInt32()) {
58          isolate->ThrowException(Exception::TypeError(
59                  String::NewFromUtf8(isolate, "Wrong arguments")));
60          return;
61      }
62
63      auto arg0 = args[0];
64      auto type = (int) args[1]->IntegerValue(); // queue type
65      auto m = (size_t) args[2]->IntegerValue(); // m
66      auto n = (size_t) args[3]->IntegerValue(); // n
67      auto size = m * n;
68      auto times = (size_t) args[4]->IntegerValue(); // times
69
70      auto buf = (int32_t *) Buffer::Data(arg0);
71      auto len = Buffer::Length(arg0) / sizeof(int32_t);
72
73      if (size * times > len) {
74          stringstream errStr;
75          errStr << "Buffer too small, need: " << size * times << ", current: " <<
            ↪  len;
76          isolate->ThrowException(Exception::TypeError(
77                  String::NewFromUtf8(isolate, errStr.str().c_str())));
78          return;
79      }
80
81      type = max(0, min(3, type));
82
83
84  //    cout << arg1 << "\t" << len << "\t";
85
86      long time = 0;
87      for (size_t i = 0; i < times; i++, buf += size) {
88          time += path_test((size_t) type, m, n, buf);
89      }
```

```
    args.GetReturnValue().Set(Integer::New(isolate, (int32_t) (time)));
}

void GetClocksPerSec(const FunctionCallbackInfo<Value> &args) {
    Isolate *isolate = args.GetIsolate();
    args.GetReturnValue().Set(Integer::New(isolate, CLOCKS_PER_SEC));
}

void init(Local<Object> exports) {
    NODE_SET_METHOD(exports, "generate", Generate);
    NODE_SET_METHOD(exports, "queue", Queue);
    NODE_SET_METHOD(exports, "getClocksPerSec", GetClocksPerSec);
}

NODE_MODULE(queue, init);
```

3.2.4 binding.gyp

```
{
    "targets": [
        {
            "target_name": "queue",
            "sources": [ "queue_wrapper.cpp" ],
        }
    ]
}
```

3.2.5 benchmark.js

```
const fs = require('fs');
const path = require('path');
const addon = require('./build/Release/queue');
const gauge = require('gauge');
const bar = new gauge(process.stderr, {
    updateInterval: 1,
    cleanupOnExit: true
});
bar.show();

const SIZE = 1e8;
const EXP_MAX = 4;
const buf = addon.generate("test", SIZE);
const CLOCKS_PER_SEC = addon.getClocksPerSec();
const MAX_TIME = 1 * CLOCKS_PER_SEC;

const ALGORITHM_MAX = 4;
const ALGORITHM_NAME = [
    "binary",
    "unsorted",
    "fibonacci",
```

```
    "cpp_standard"
];
const ALGORITHM_ACTIVE = [];
let result = [];
for (let i = 0; i < ALGORITHM_MAX; i++) {
    ALGORITHM_ACTIVE.push(true);
    result.push(null);
}


const REPEAT_TIMES = [100, 10, 2, 1, 1, 1, 1];
const PARTITION_ARR = [100, 100, 50, 50, 20, 20, 20];
const WEIGHT_ARR = require('./progress.json');
let total_time = [0, 0, 0, 0, 0, 0, 0];


let tasks = [];
let base = 1;
let weight_all = 0;


for (let exp = 0; exp < EXP_MAX; exp++) {
    let size = base;
    let partition = PARTITION_ARR[exp];
    base *= 10;
    for (let mul = 1; mul < partition - 1 && size < base; mul++) {
        //console.log(size, base, Math.ceil(base / partition ));
        for (let i = 0; i < ALGORITHM_MAX; i++) {
            let weight = WEIGHT_ARR[exp] || 1;
            weight_all += weight;
            tasks.push({
                size: size,
                order: i,
                times: REPEAT_TIMES[exp],
                weight: weight,
                exp: exp
            });
        }
        size += Math.ceil(base / partition);
    }
}


let queue = [];
let progress = 0;

tasks.forEach((value) => {
    queue.push(() => {
        progress += 1 / weight_all * value.weight;
```

```
71
72          if (!ALGORITHM_ACTIVE[value.order]) {
73              result[value.order] = null;
74              return [value, -1];
75          }
76
77          const newBuf = Buffer.from(buf.slice(0, value.size * value.size *
            ↪  value.times * 4));
78          const totalTime = addon.queue(newBuf, value.order, value.size,
            ↪  value.size, value.times);
79          const averageTime = totalTime / value.times;
80          total_time[value.exp] += totalTime;
81
82          result[value.order] = newBuf;
83          if (averageTime > MAX_TIME) {
84              ALGORITHM_ACTIVE[value.order] = false;
85              //console.log(value.order);
86          }
87
88          /*if (value.order === ALGORITHM_MAX - 1) {
89              for (let i = 0; i < value.order; i++) {
90                  const temp = result[i];
91                  if (temp && Buffer.compare(temp, result[value.order]) !== 0) {
92                      //console.error(value.size, ALGORITHM_NAME[i]);
93                  }
94              }
95          }*/
96
97          return [value, averageTime];
98      });
99  });
100
101 const file = fs.openSync(path.resolve(__dirname, 'result'), 'w');
102
103
104 const func = () => {
105
106     const [data, averageTime] = (queue.shift())();
107
108     if (averageTime > 0) {
109         const time = Math.round(averageTime) / CLOCKS_PER_SEC;
110         const blanks = "                    ";
111         console.log(`size: ${data.size}, algorithm:
            ↪  ${ALGORITHM_NAME[data.order]}, time: ${time}s ${blanks}`);
112         fs.writeSync(file, `${data.size} ${data.order} ${averageTime /
            ↪  CLOCKS_PER_SEC}\n`);
113     }
114
115     if (tasks.length) {
```

```
116             const task = tasks.shift();
117             bar.show(`${Math.round(progress * 100)}%`, progress);
118             bar.pulse(`size: ${task.size}, algorithm:
                ↪ ${ALGORITHM_NAME[task.order]}`);
119         }
120
121         if (queue.length) {
122             setTimeout(func, 0);
123         } else {
124             fs.closeSync(file);
125             let data = [];
126             total_time.forEach((value) => {
127                 const ratio = Math.round(value / total_time[0]);
128                 data.push(ratio);
129                 //console.log(ratio);
130             });
131             fs.writeFileSync(path.resolve(__dirname, 'progress.json'),
                ↪ JSON.stringify(data));
132         }
133     };
134
135     tasks.shift();
136     func();
```

3.2.6 benchmark.m

```
1   fid = fopen('result', 'r');
2   tline = fgetl(fid);
3   data = [];
4   while ischar(tline)
5       A = sscanf(tline, '%d %d %f');
6       data = [data; A'];
7       tline = fgetl(fid);
8   end
9   fclose(fid);
10
11
12  figure(1);
13  clf;
14
15  hold on;
16  for i=0:3
17      subdata = data(data(:,2)==i,[1 3]);
18      plot(subdata(:,1),subdata(:,2));
19  end
20  hold off;
21
22  set(gca,'XScale','log');
23  set(gca,'YScale','log');
24  axis([1 2e3 0 1]);
```

```
25 legend('binary heap','unsorted heap','fibonacci heap','cpp
   ↪ standard','Location','northwest');
26 set(gca,'Fontsize',20);
27 saveas(gcf,'fig1.png');
28
29 % figure(2);
30 % clf;
31 %
32 % hold on;
33 % for i=0:3
34 %     subdata = data(data(:,2)==i,[1 3]);
35 %     plot(subdata(:,1),subdata(:,2));
36 % end
37 % hold off;
38 %
39 % axis([10 400 0 2e-5]);
40 % set(gca,'XScale','log');
41 % legend('binary heap','unsorted heap','fibonacci heap','cpp
   ↪ standard','Location','northwest');
42 % set(gca,'Fontsize',20);
43 % saveas(gcf,'fig2.png');
```