

ДОДАТОК А

Результати обчислювального експерименту

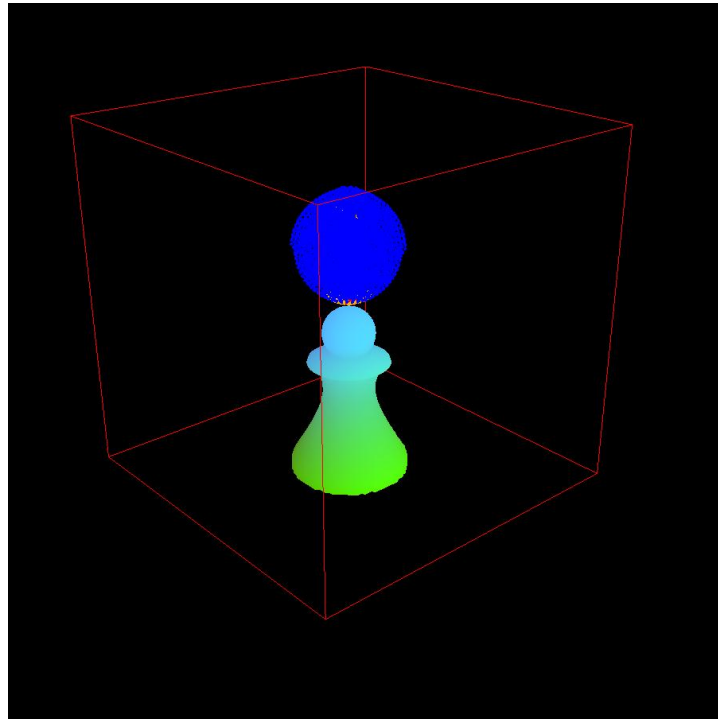


Рисунок А.1 – Симуляція течії із використанням оптимізованого алгоритму
у момент часу $t = 0$

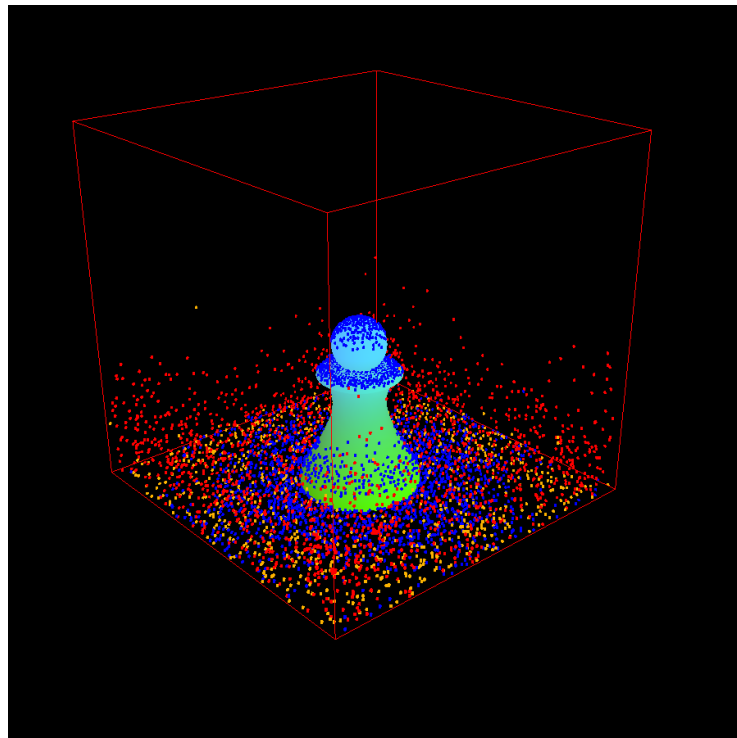


Рисунок А.2 – Симуляція течії із використанням оптимізованого алгоритму
у момент часу $t = 5$

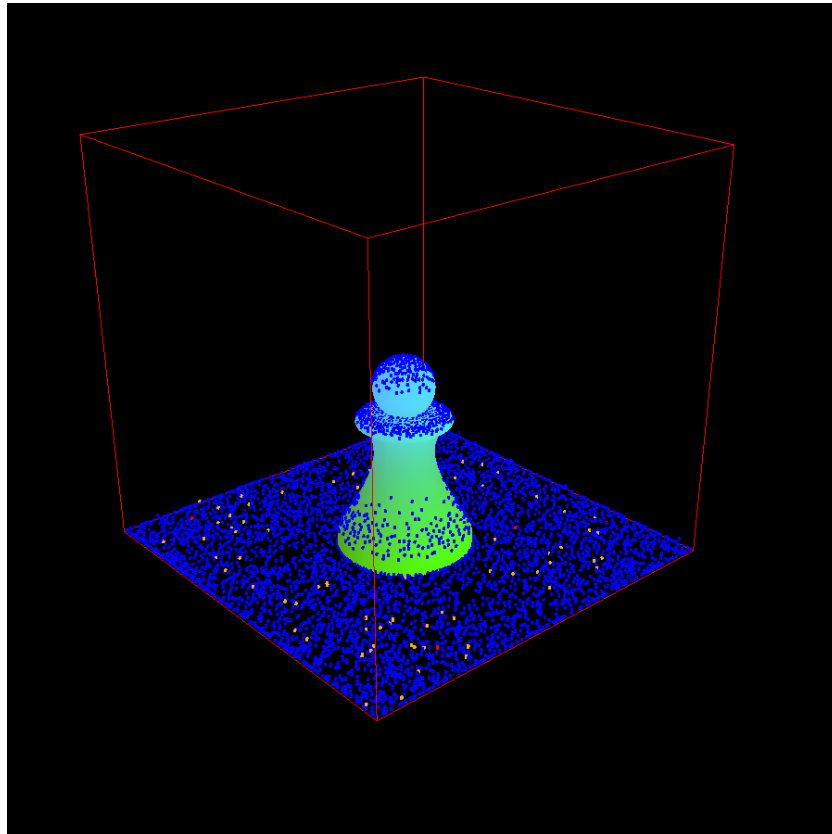


Рисунок А.3 – Симуляція течії із використанням оптимізованого алгоритму
у момент часу $t = 10$

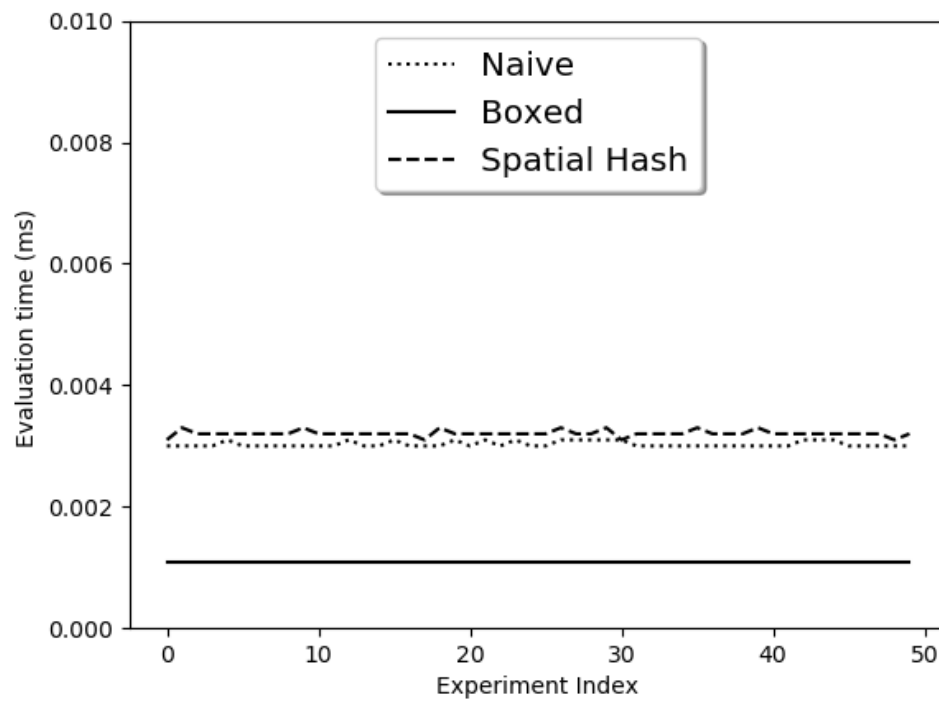


Рисунок А.4 – Порівняння середнього часу обчислення усіх алгоритмів
для $r = 8$ (50 тестів)

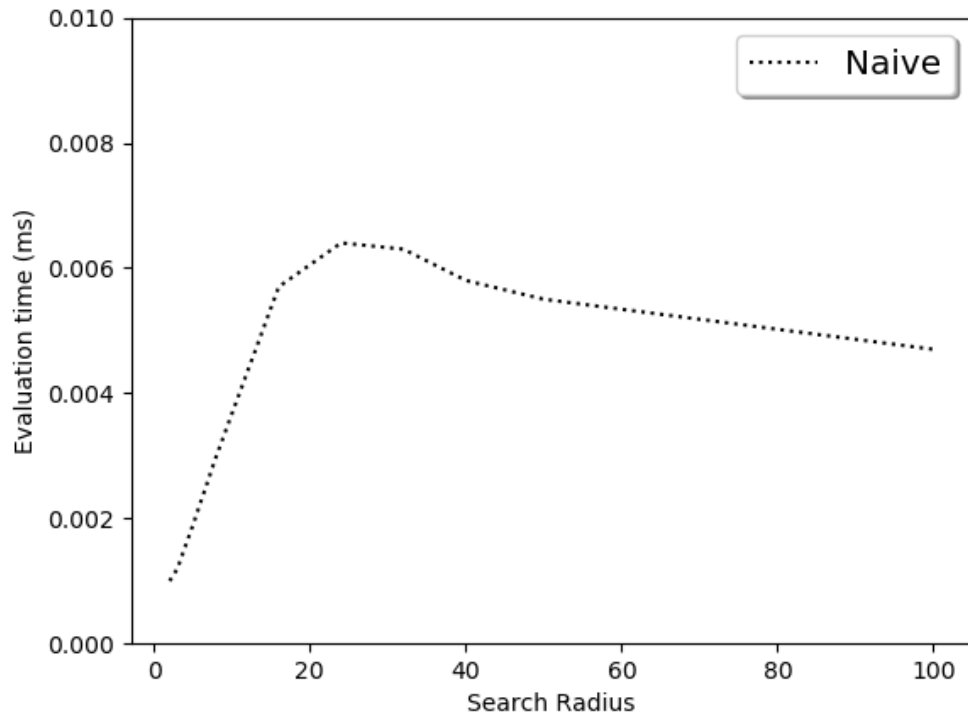


Рисунок А.5 – Середній час обчислення наївного алгоритму
для $r = 2, 3, 5, 8, 16, 24, 32, 40, 50, 100$

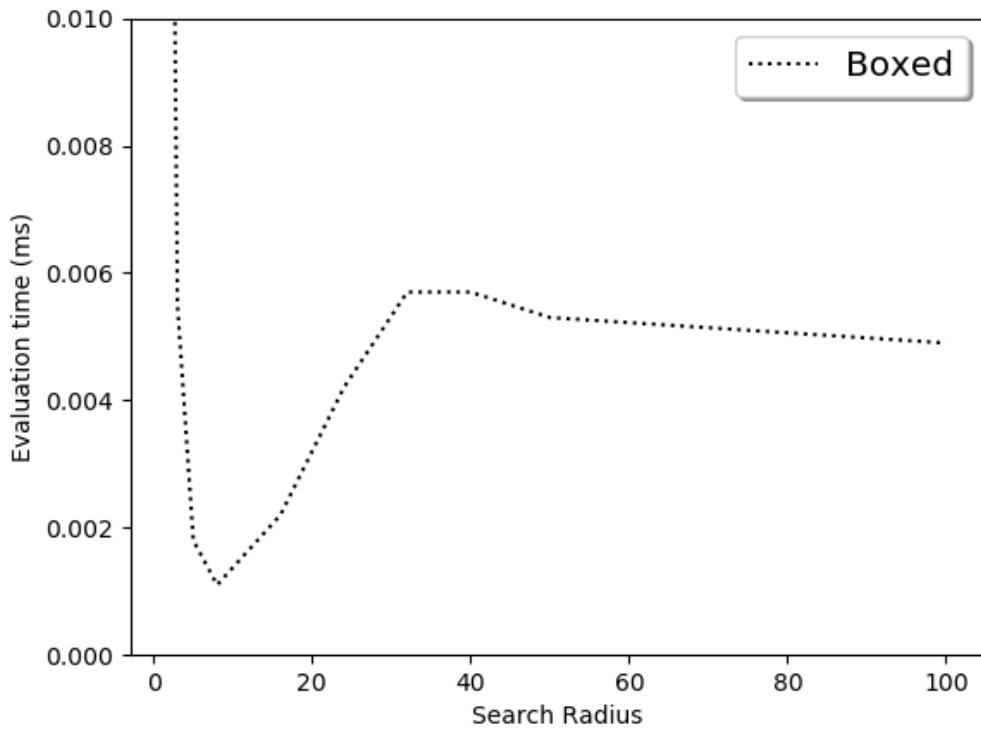


Рисунок А.6 – Середній час обчислення боксового алгоритму
для $r = 2, 3, 5, 8, 16, 24, 32, 40, 50, 100$

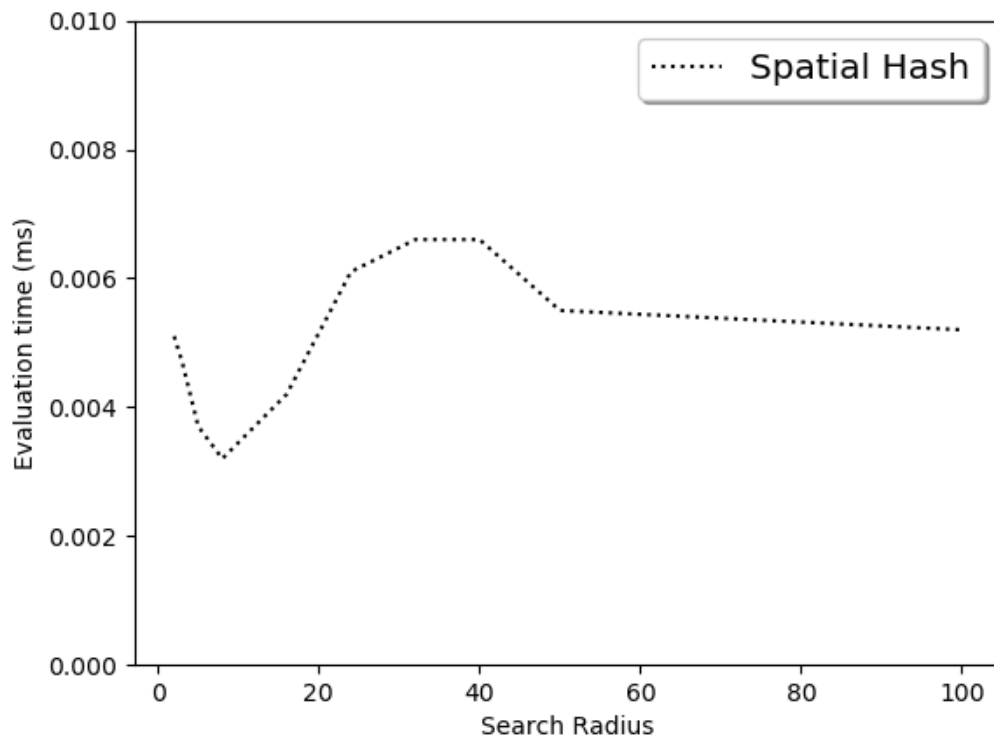


Рисунок А.7 – Середній час обчислення алгоритму просторового хешування
для $r = 2, 3, 5, 8, 16, 24, 32, 40, 50, 100$

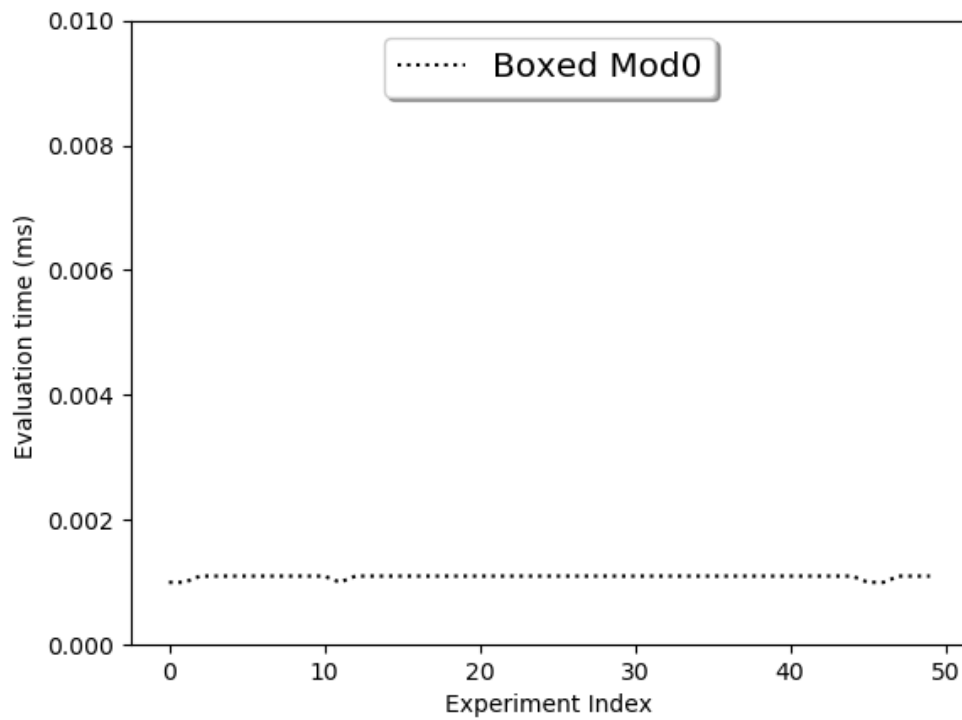


Рисунок А.8 – Час обчислення модифікації 0 боксового алгоритму
для $r = 8$ (50 тестів)

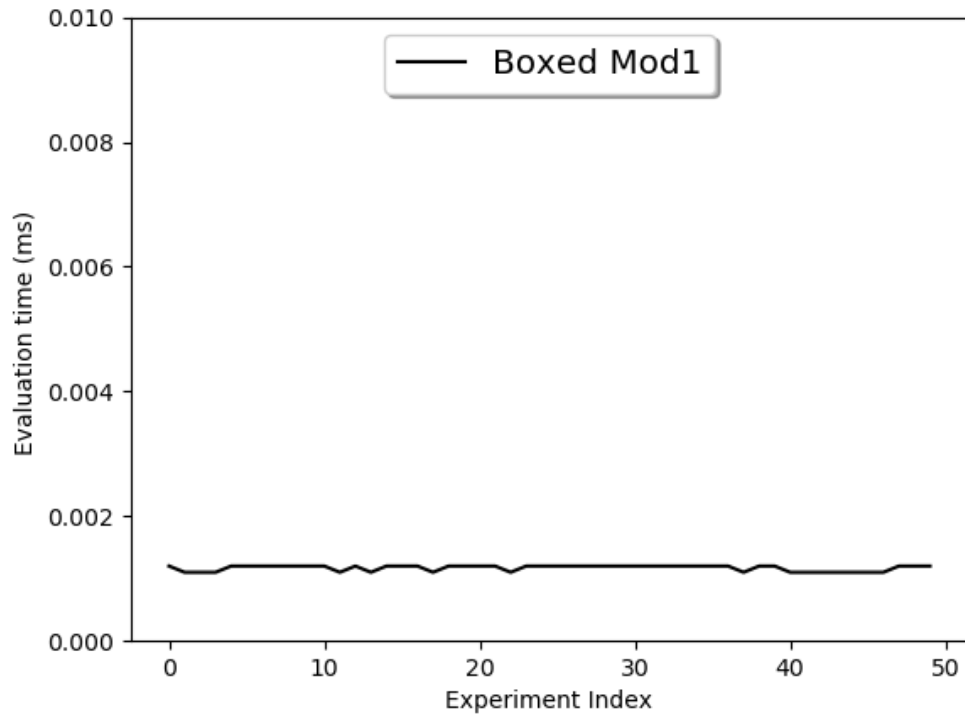


Рисунок А.9 – Час обчислення модифікації 1 боксового алгоритму
для $r = 8$ (50 тестів)

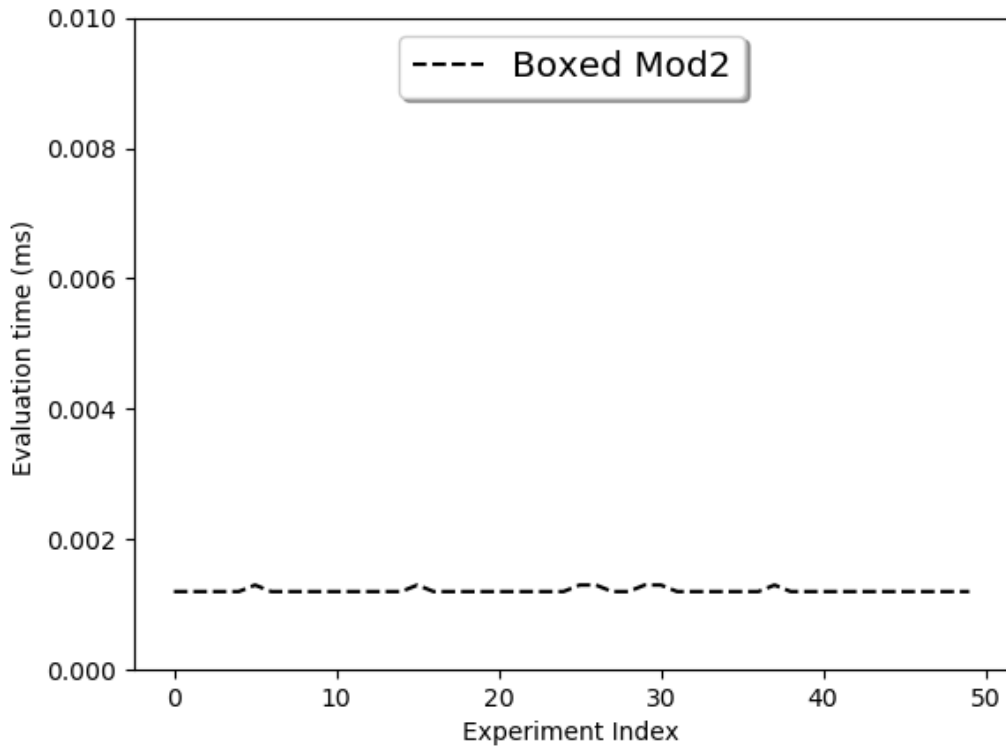


Рисунок А.10 – Час обчислення модифікації 2 боксового алгоритму
для $r = 8$ (50 тестів)

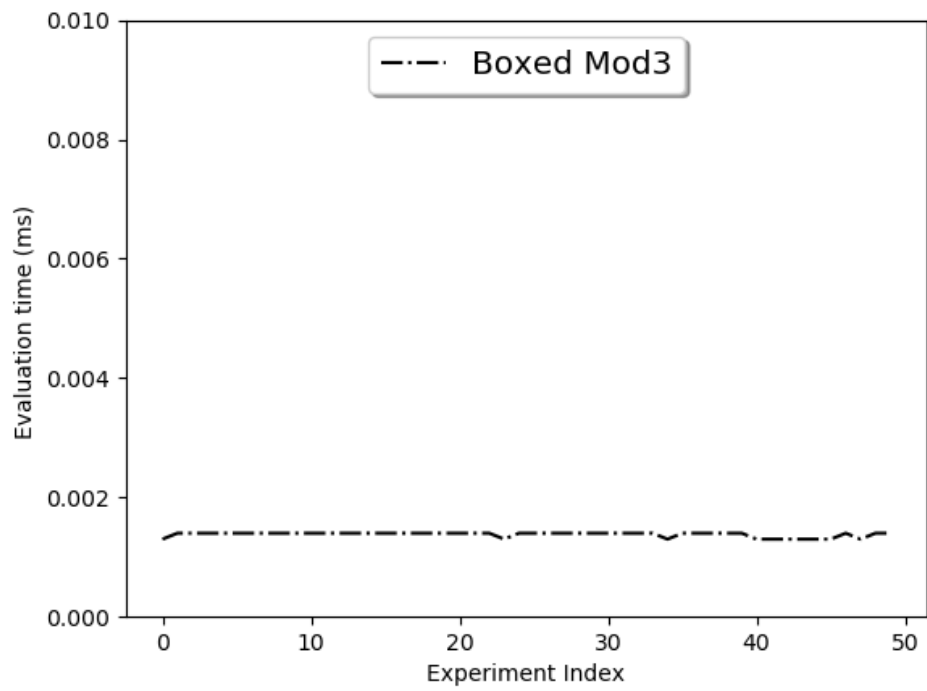


Рисунок А.11 – Час обчислення модифікації 3 боксового алгоритму
для $r = 8$ (50 тестів)

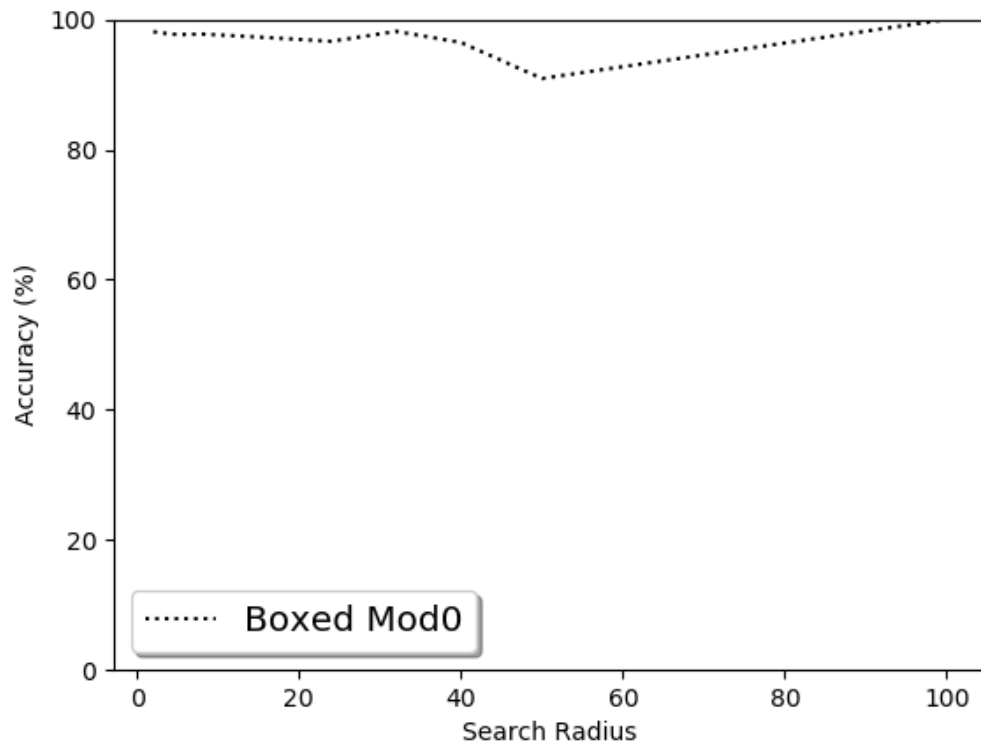


Рисунок А.12 – Точність обчислення модифікації 0 боксового алгоритму
для $r = 8$ (50 тестів)

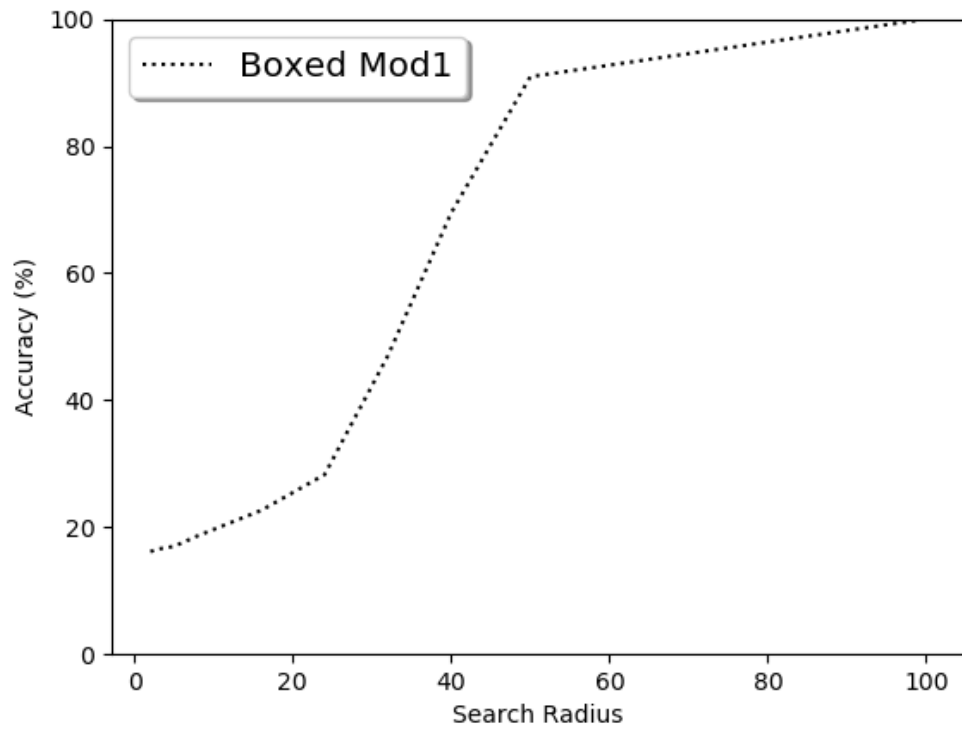


Рисунок А.13 – Точність обчислення модифікації 1 боксового алгоритму
для $r = 8$ (50 тестів)

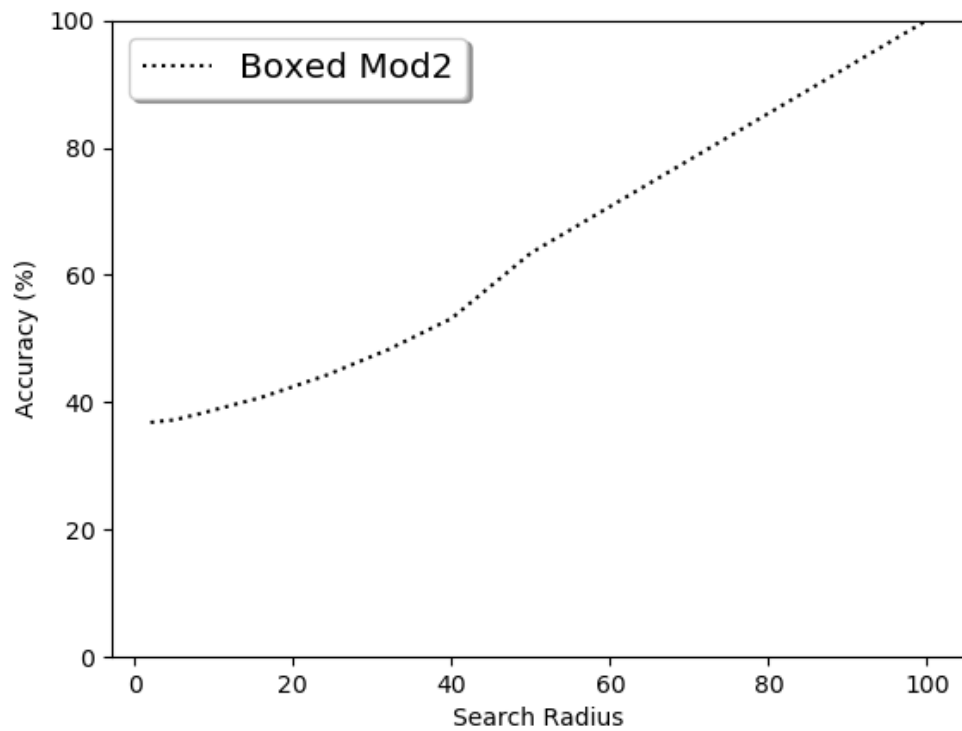


Рисунок А.14 – Точність обчислення модифікації 2 боксового алгоритму
для $r = 8$ (50 тестів)

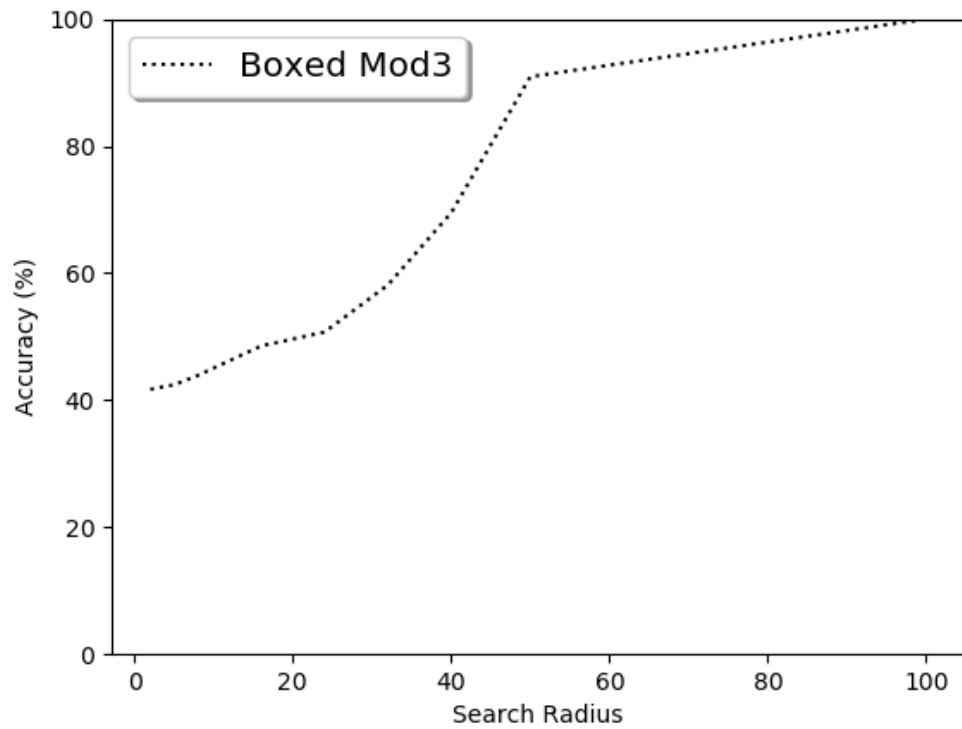


Рисунок А.15 – Точність обчислення модифікації 3 боксового алгоритму
для $r = 8$ (50 тестів)

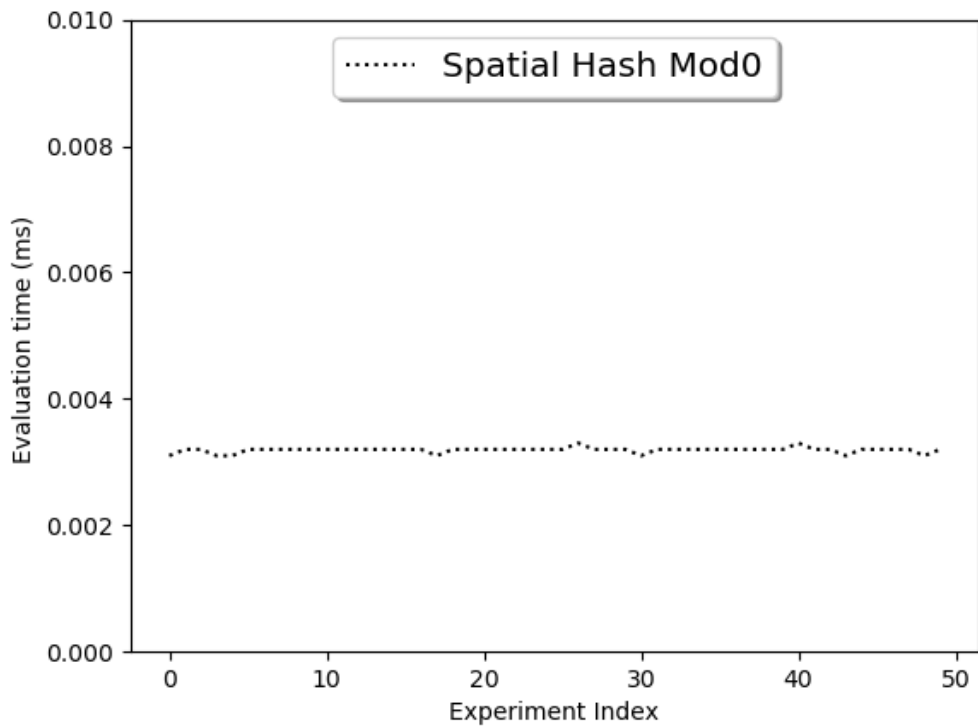


Рисунок А.16 – Час обчислення модифікації 0 алгоритму
просторового хешування для $r = 8$ (50 тестів)

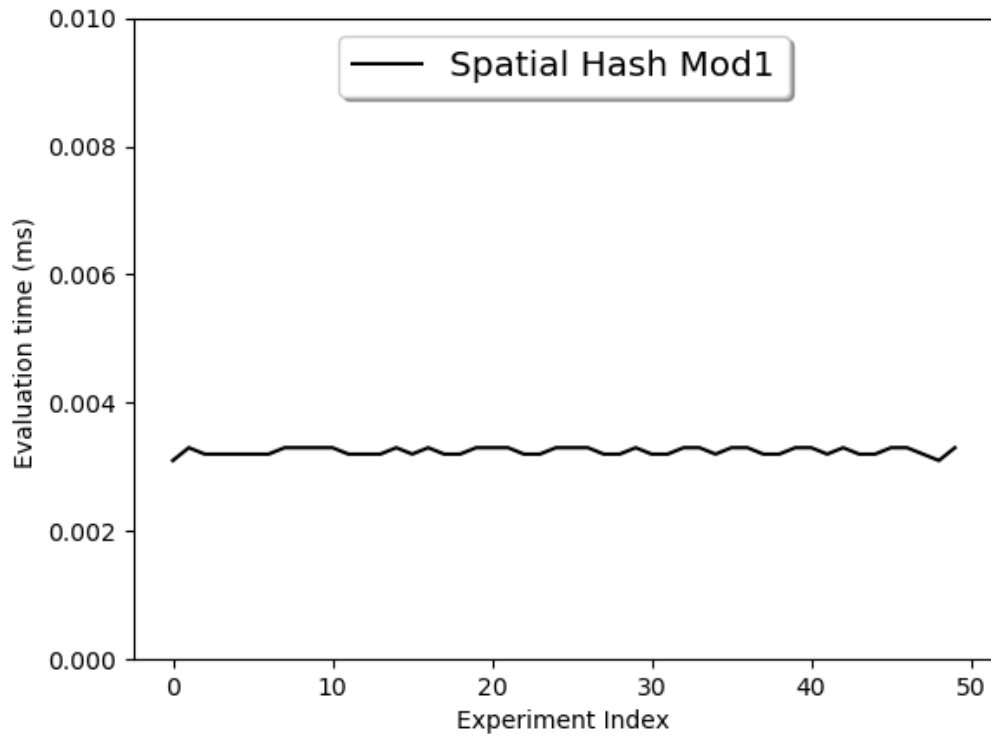


Рисунок А.17 – Час обчислення модифікації 1 алгоритму просторового хешування для $r = 8$ (50 тестів)

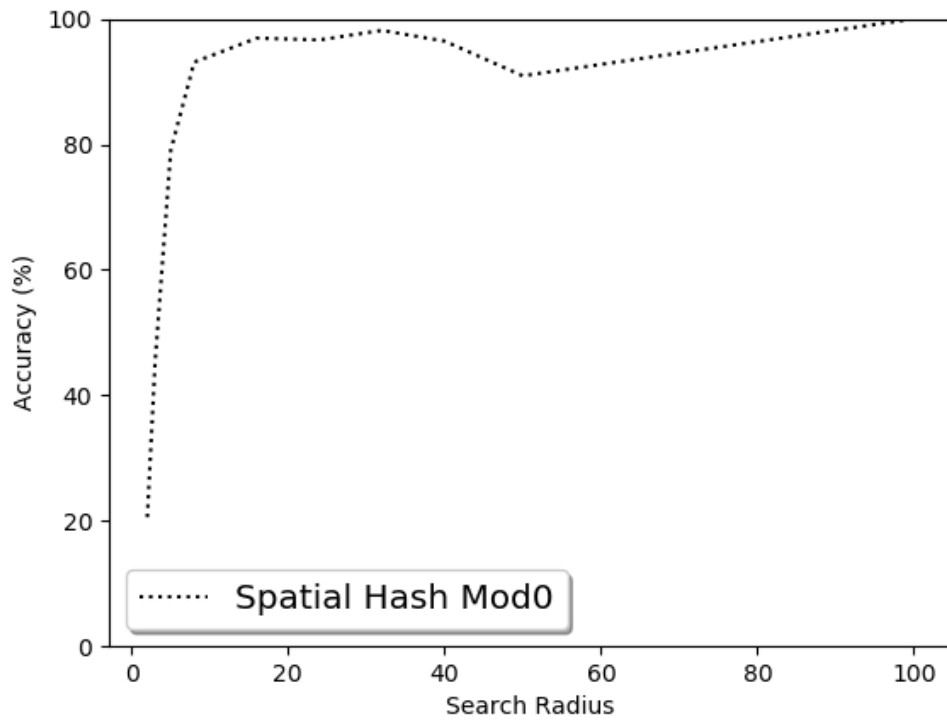


Рисунок А.18 – Точність обчислення модифікації 0 алгоритму просторового хешування для $r = 8$ (50 тестів)

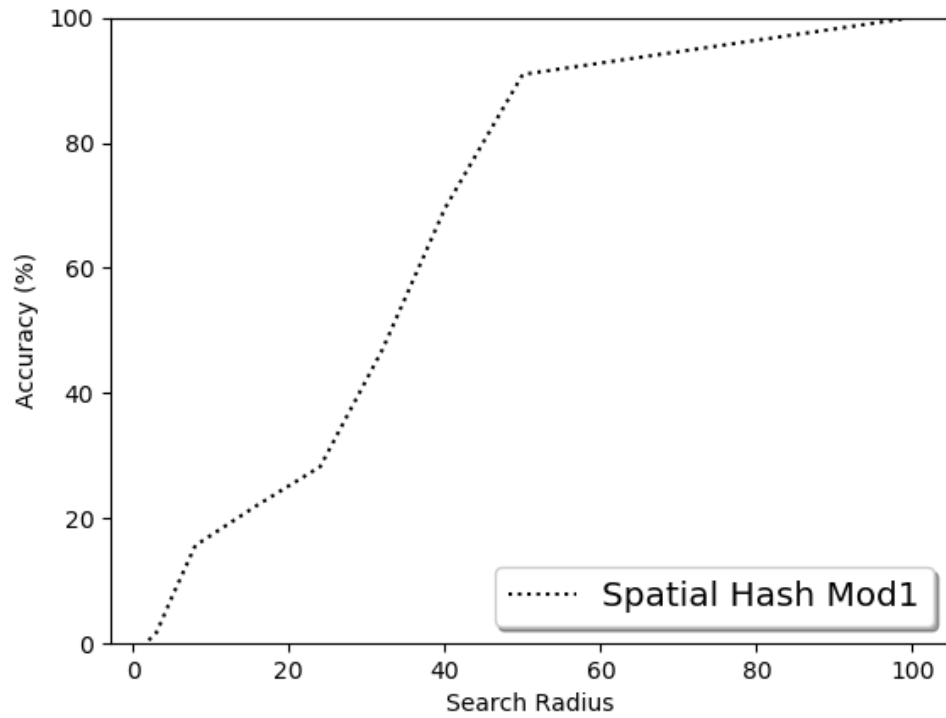


Рисунок А.19 – Точність обчислення модифікації 1 алгоритму просторового хешування для $r = 8$ (50 тестів)

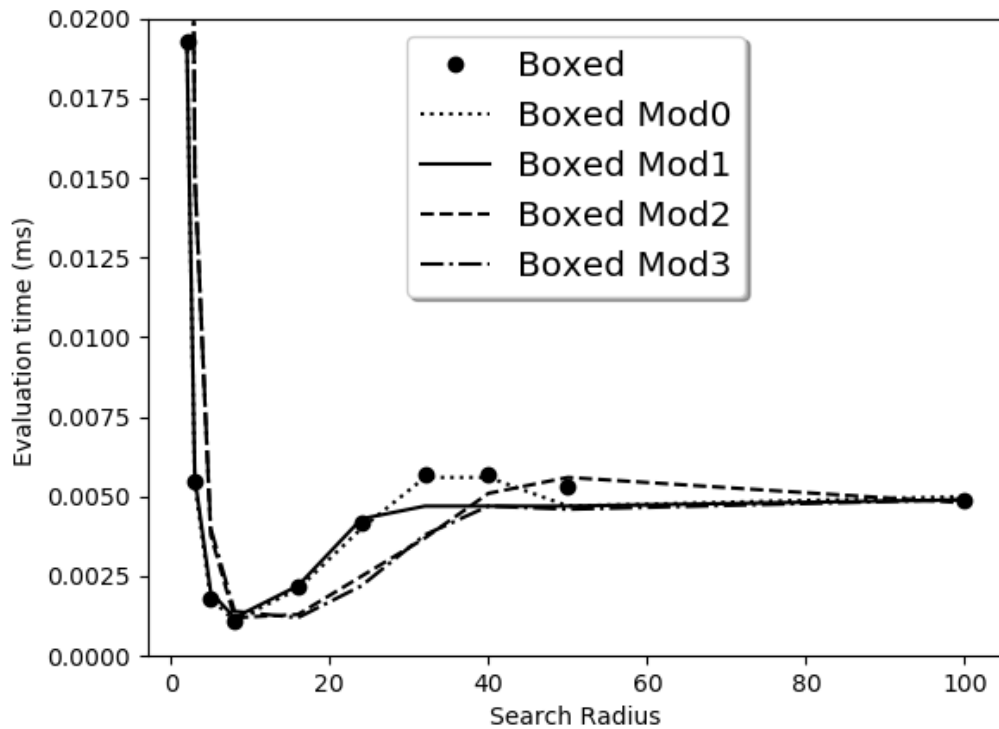


Рисунок А.20 – Порівняння середнього часу обчислень модифікацій боксового алгоритму для $r = 2, 3, 5, 8, 16, 24, 32, 40, 50, 100$

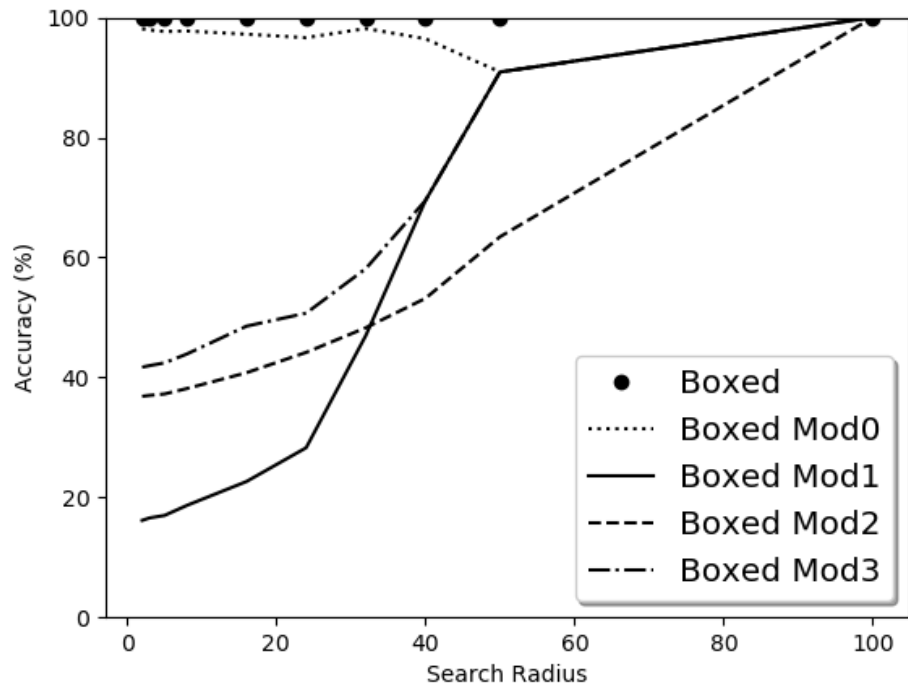


Рисунок А.21 – Порівняння середньої точності обчислень модифікацій боксового алгоритму для $r = 2, 3, 5, 8, 16, 24, 32, 40, 50, 100$

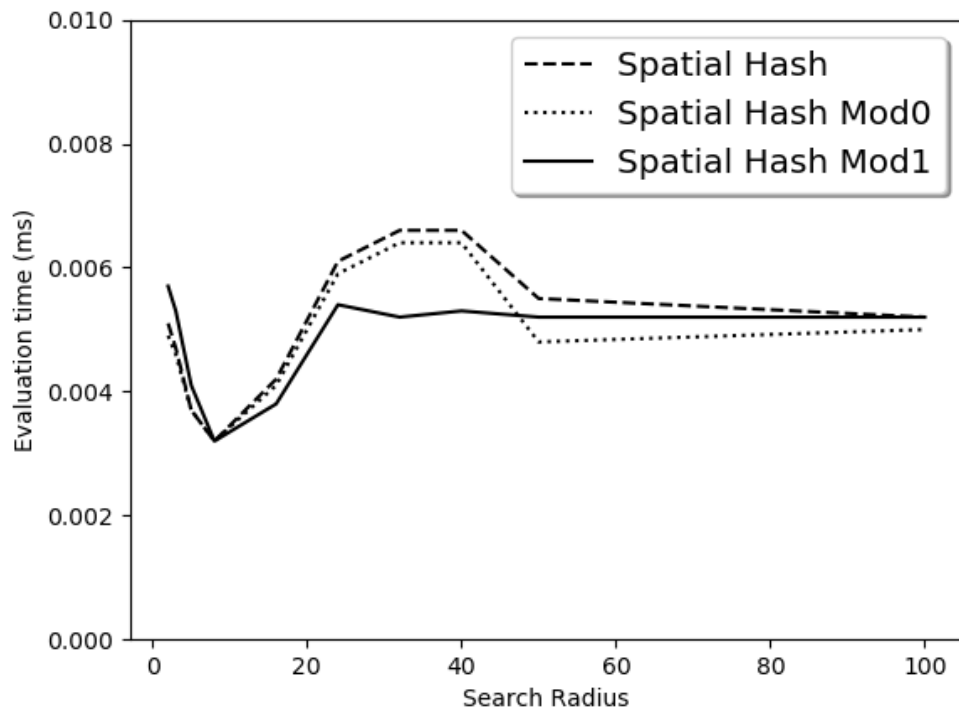


Рисунок А.22 – Порівняння середнього часу обчислень модифікації алгоритму просторового хешування для $r = 2, 3, 5, 8, 16, 24, 32, 40, 50, 100$

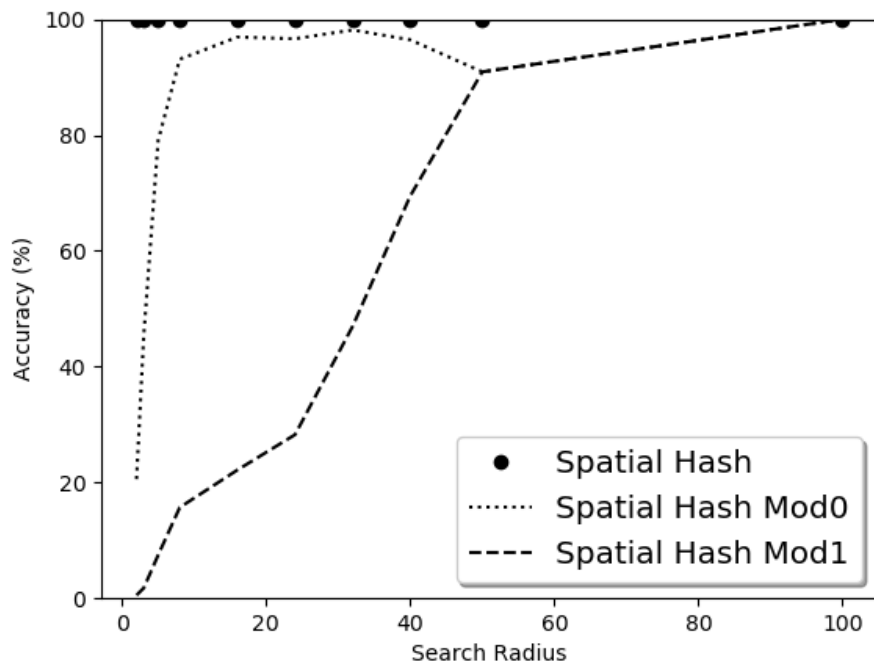


Рисунок А.23 – Порівняння середньої точності обчислень модифікацій алгоритму просторового хешування для $r = 2, 3, 5, 8, 16, 24, 32, 40, 50, 100$

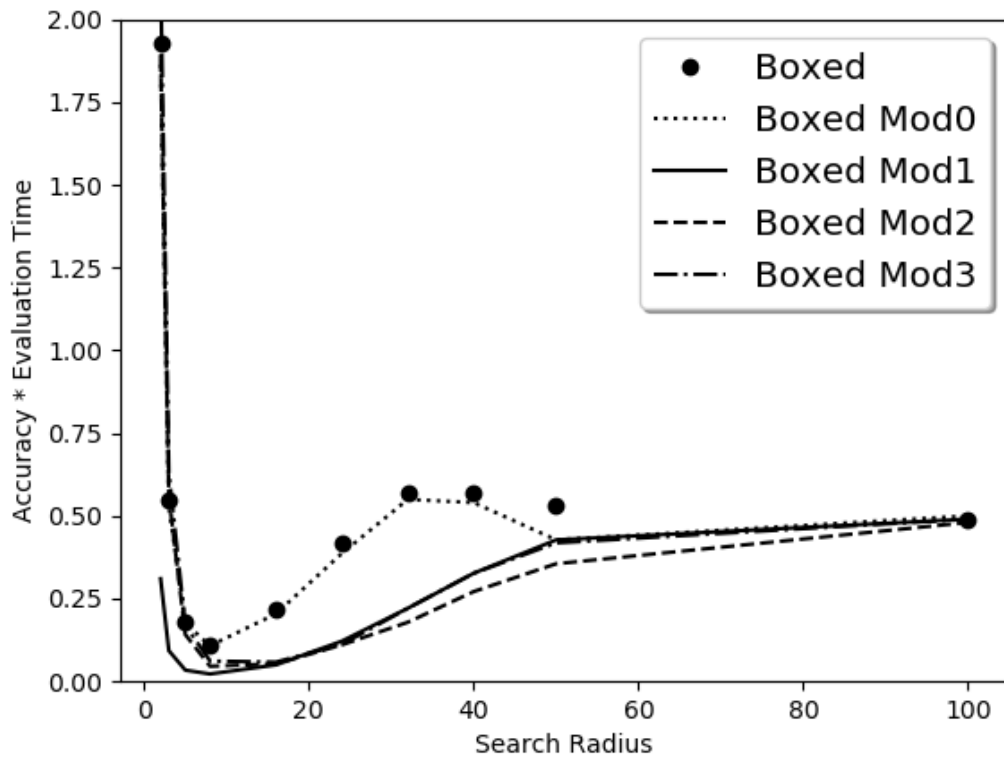


Рисунок А.24 – Точність-Час обчислення модифікацій боксового алгоритму для $r = 2, 3, 5, 8, 16, 24, 32, 40, 50, 100$

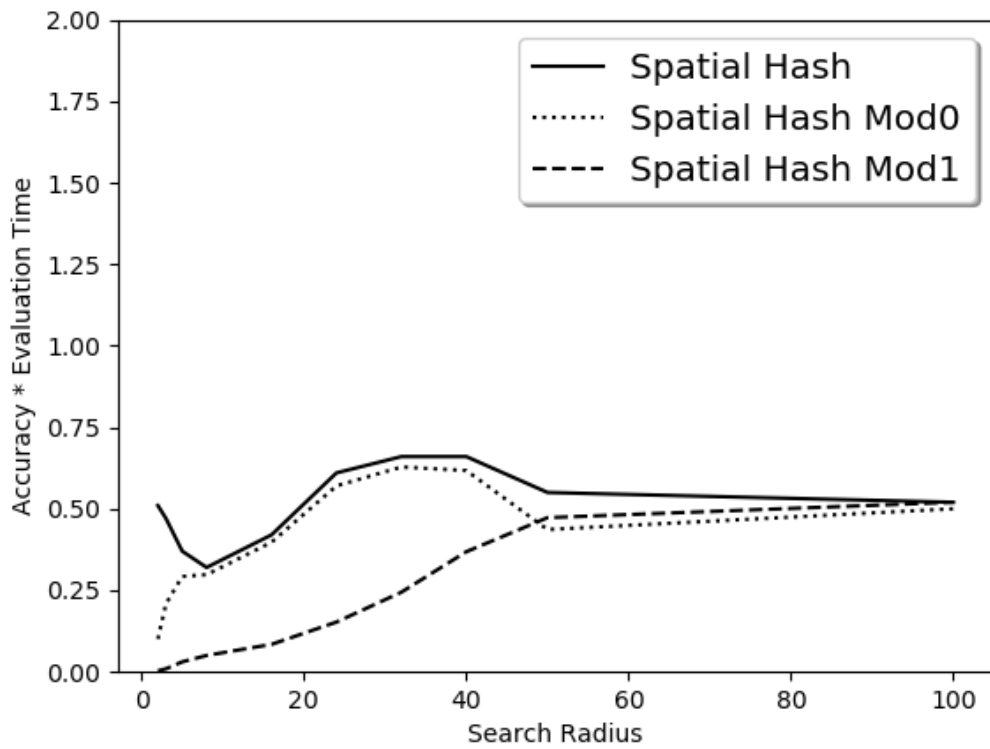


Рисунок А.25 – Точність-Час обчислення модифікацій алгоритму просторового хешування для $r = 2, 3, 5, 8, 16, 24, 32, 40, 50, 100$

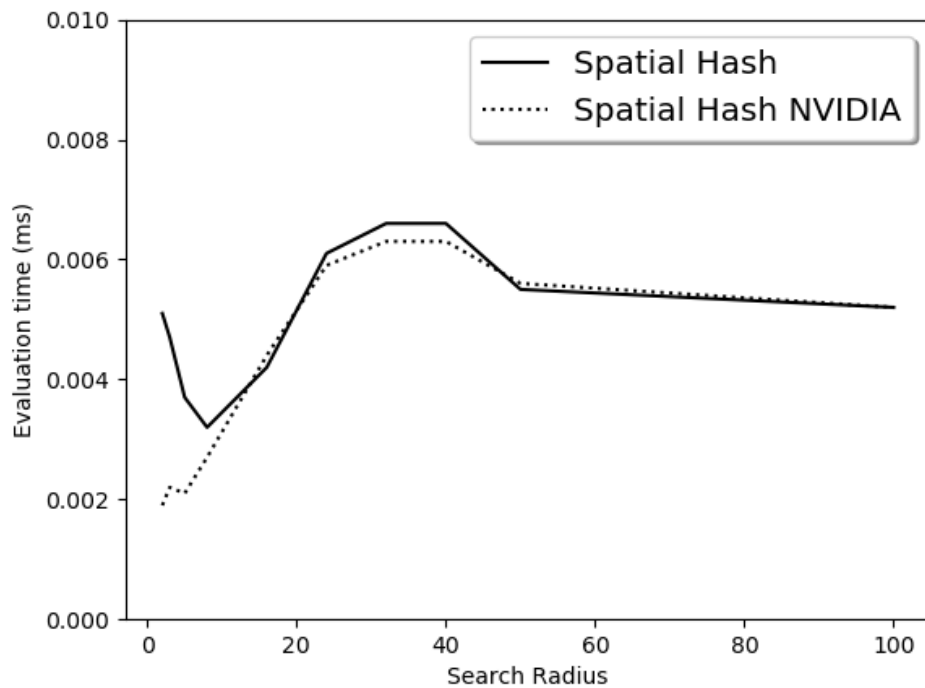


Рисунок А.26 – Порівняння часу обчислення різних хеш-функцій для алгоритму просторового хешування для $r = 2, 3, 5, 8, 16, 24, 32, 40, 50, 100$

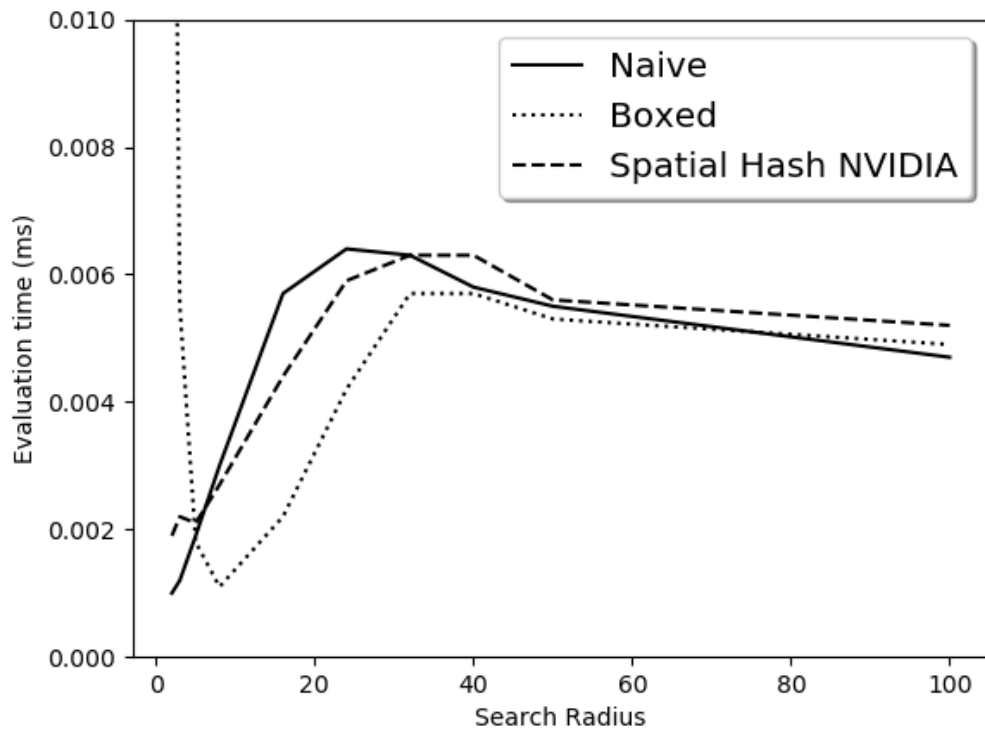


Рисунок А.27 – Порівняння часу обчислення наївного, боксового та алгоритму просторового хешування (хеш-функція NVIDIA) для $r = 2, 3, 5, 8, 16, 24, 32, 40, 50, 100$

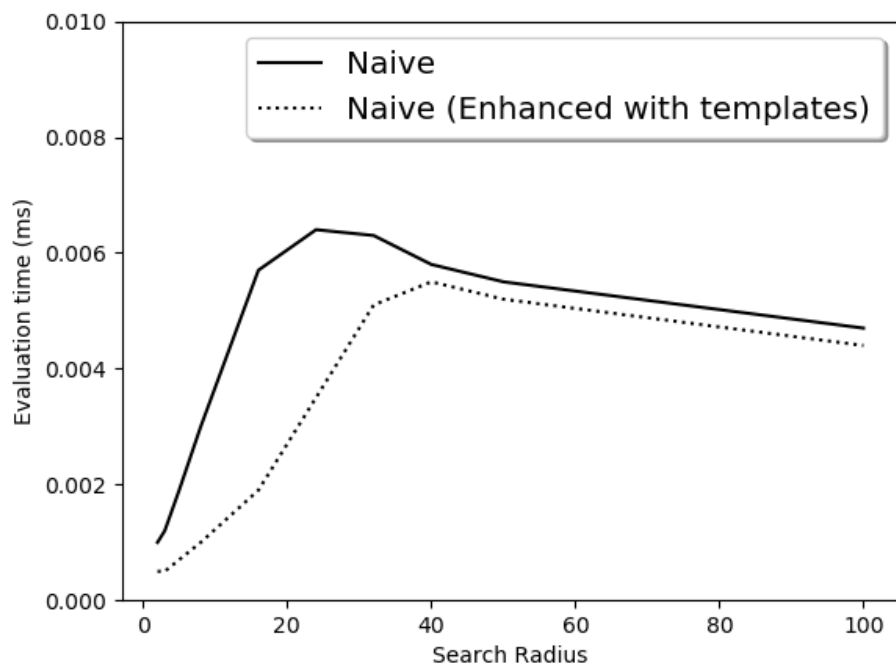


Рисунок А.28 – Порівняння часу обчислення наївного алгоритму з та без застосування шаблонів для $r = 2, 3, 5, 8, 16, 24, 32, 40, 50, 100$

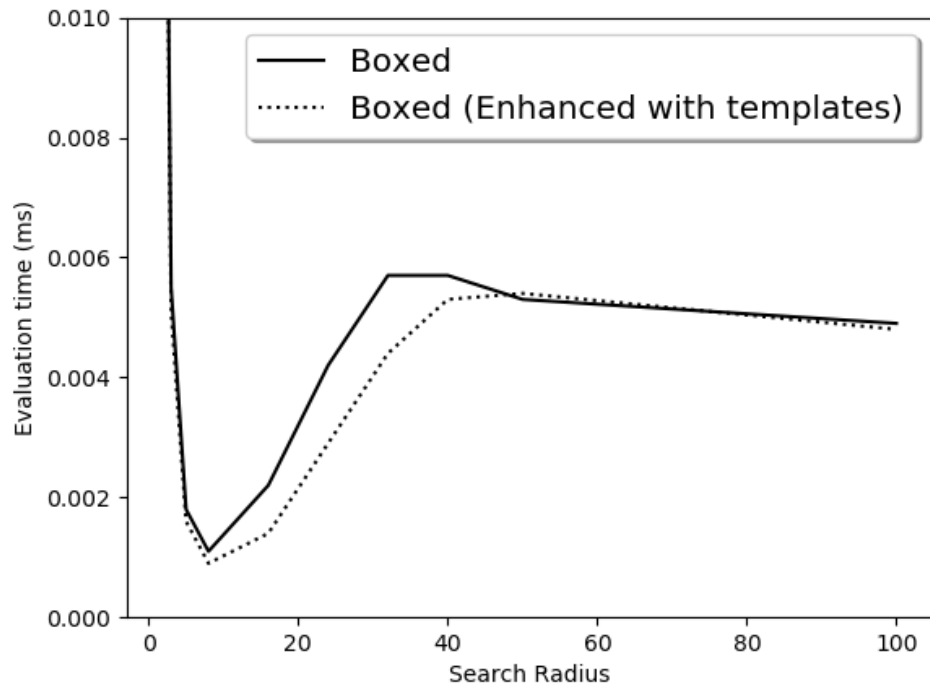


Рисунок А.29 – Порівняння часу обчислення боксового алгоритму з та без застосування шаблонів для $r = 2, 3, 5, 8, 16, 24, 32, 40, 50, 100$

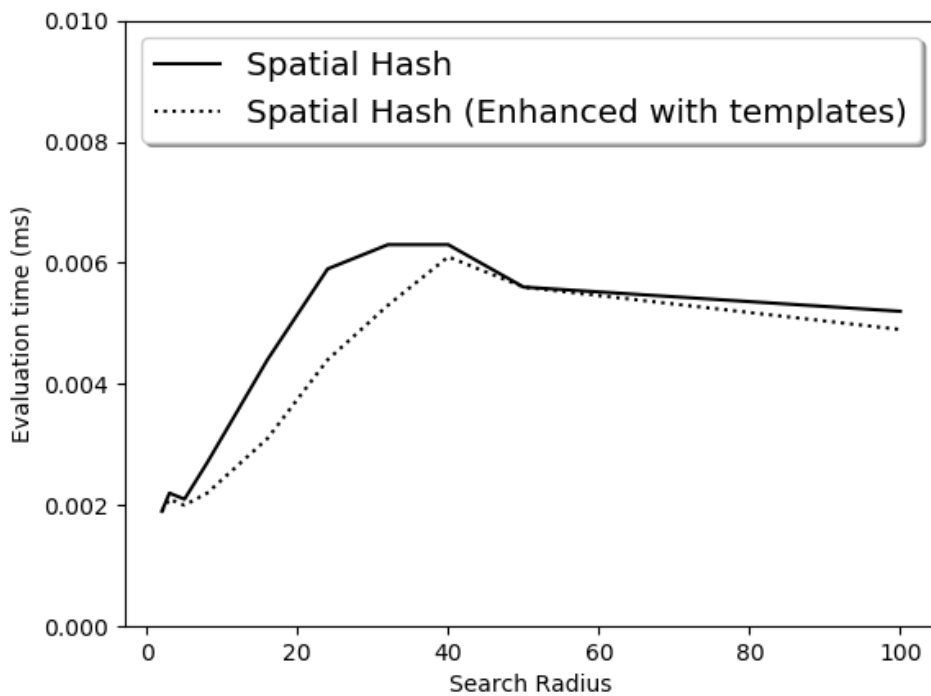


Рисунок А.30 – Порівняння часу обчислення алгоритму просторового хешування (хеш-функція з та без застосування шаблонів для $r = 2, 3, 5, 8, 16, 24, 32, 40, 50, 100$

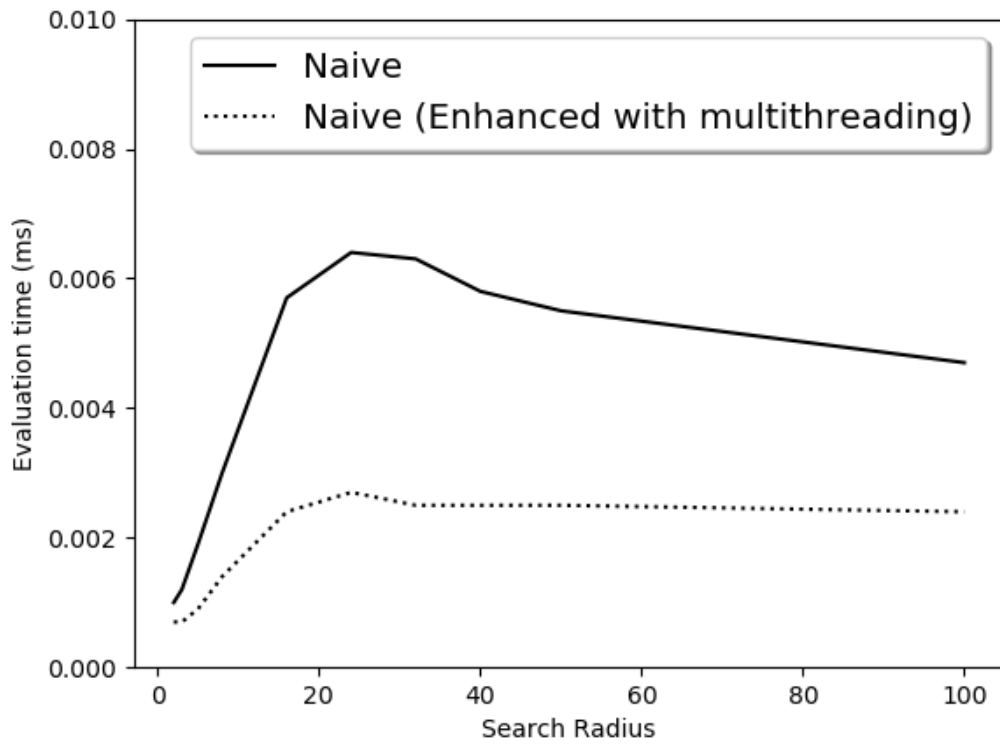


Рисунок А.31 – Порівняння часу обчислення наївного алгоритму з та без застосування багатонитковості для $r = 2, 3, 5, 8, 16, 24, 32, 40, 50, 100$

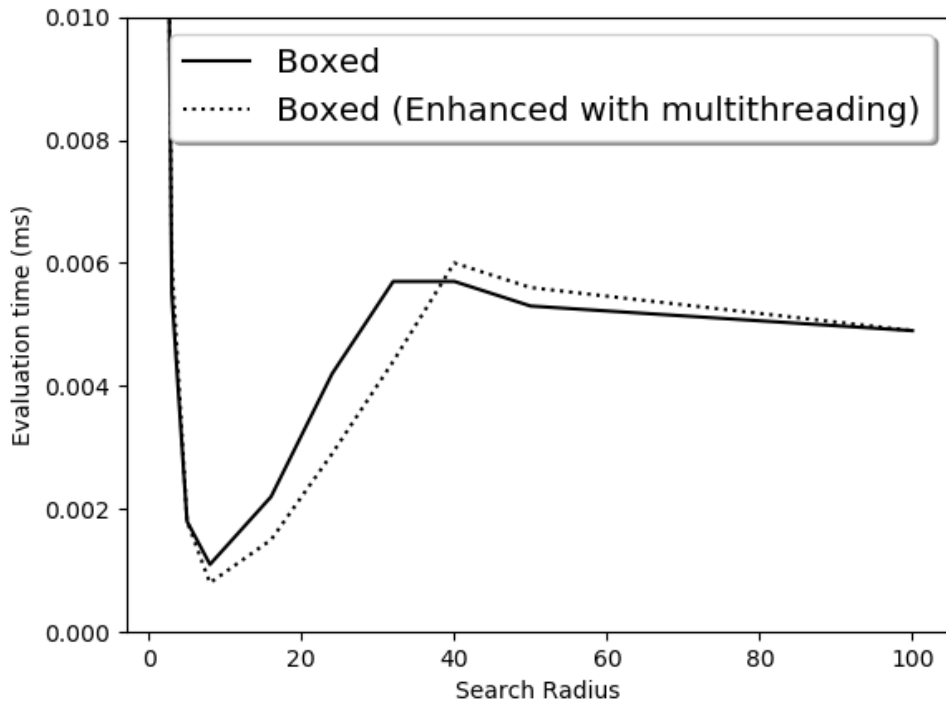


Рисунок А.32 – Порівняння часу обчислення боксового алгоритму з та без застосування багатонитковості для $r = 2, 3, 5, 8, 16, 24, 32, 40, 50, 100$

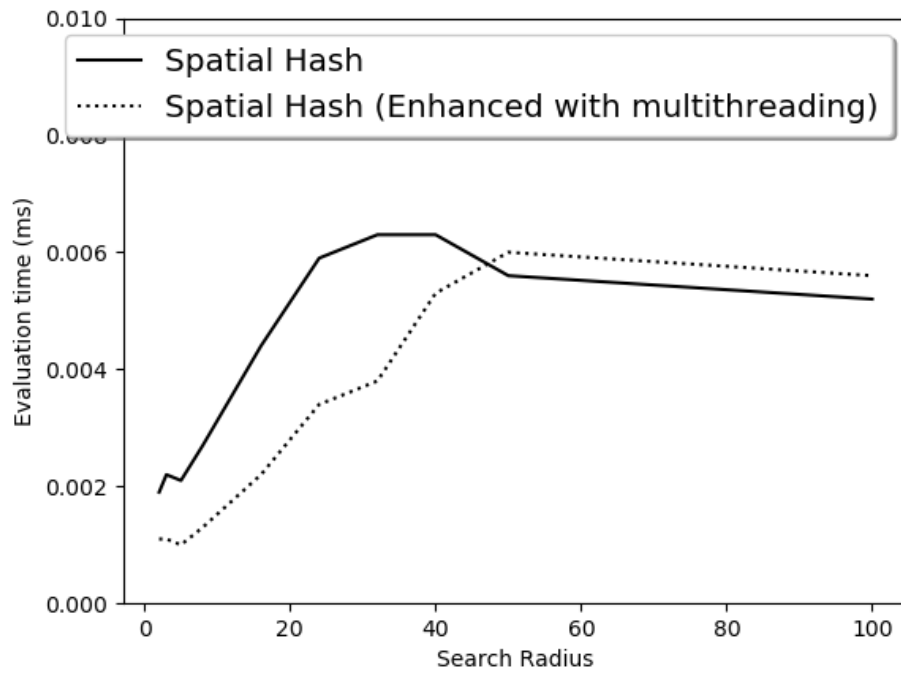


Рисунок А.33 – Порівняння часу обчислення алгоритму просторового хешування (хеш-функція NVIDIA) з та без застосування багатонитковості для $r = 2, 3, 5, 8, 16, 24, 32, 40, 50, 100$

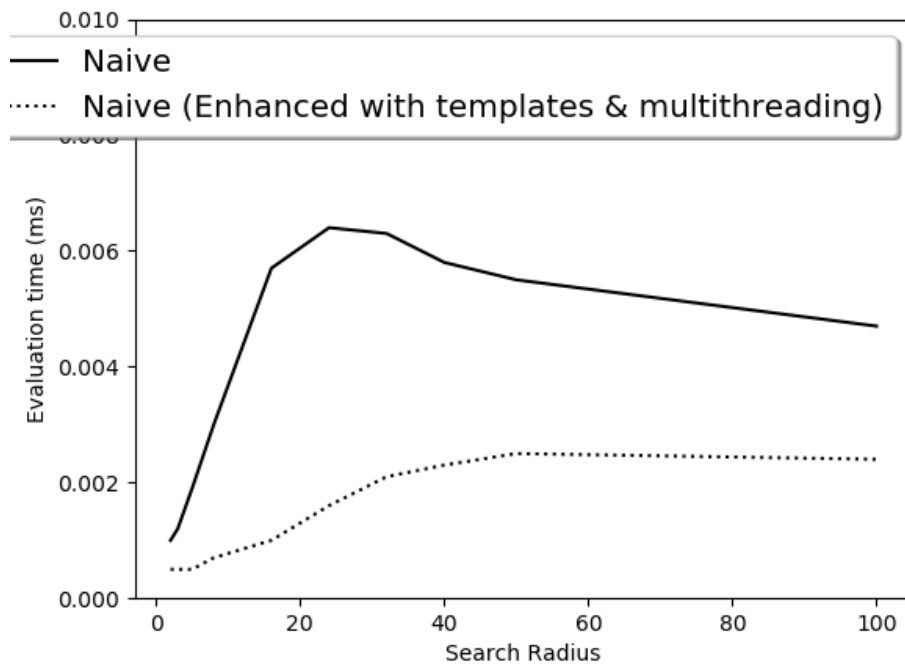


Рисунок А.34 – Порівняння часу обчислення наївного алгоритму із застосуванням шаблонів та багатонитковості та без для $r = 2, 3, 5, 8, 16, 24, 32, 40, 50, 100$

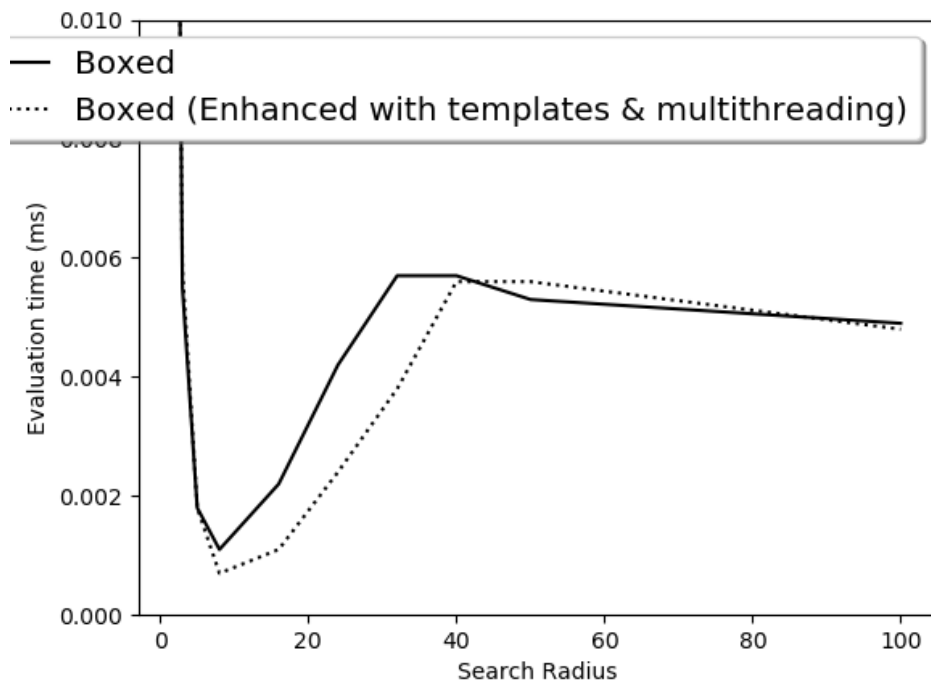


Рисунок А.35 – Порівняння часу обчислення боксового алгоритму із застосуванням шаблонів та багатонитковості та без для $r = 2, 3, 5, 8, 16, 24, 32, 40, 50, 100$

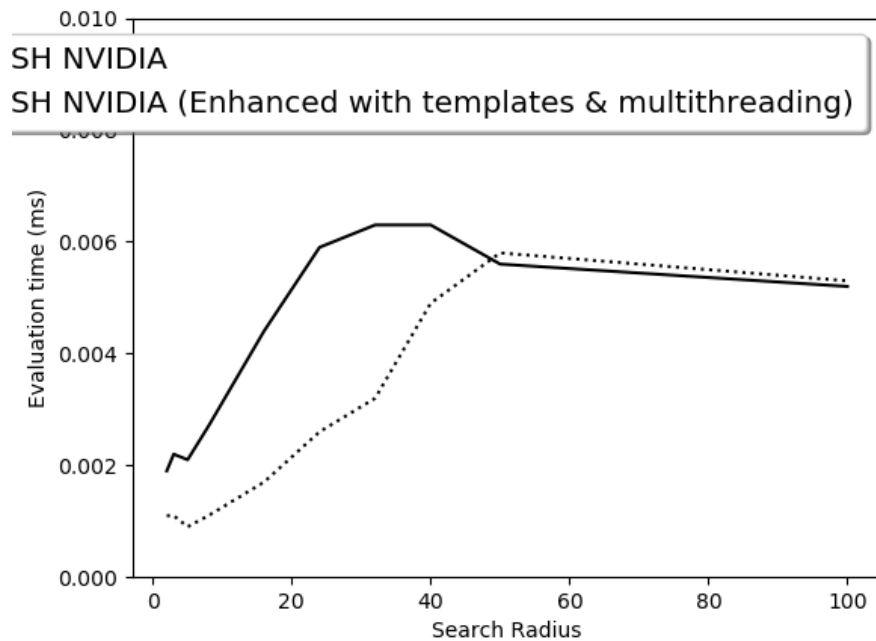


Рисунок А.36 – Порівняння часу обчислення алгоритму просторового хешування із застосуванням шаблонів та багатонитковості та без для $r = 2, 3, 5, 8, 16, 24, 32, 40, 50, 100$

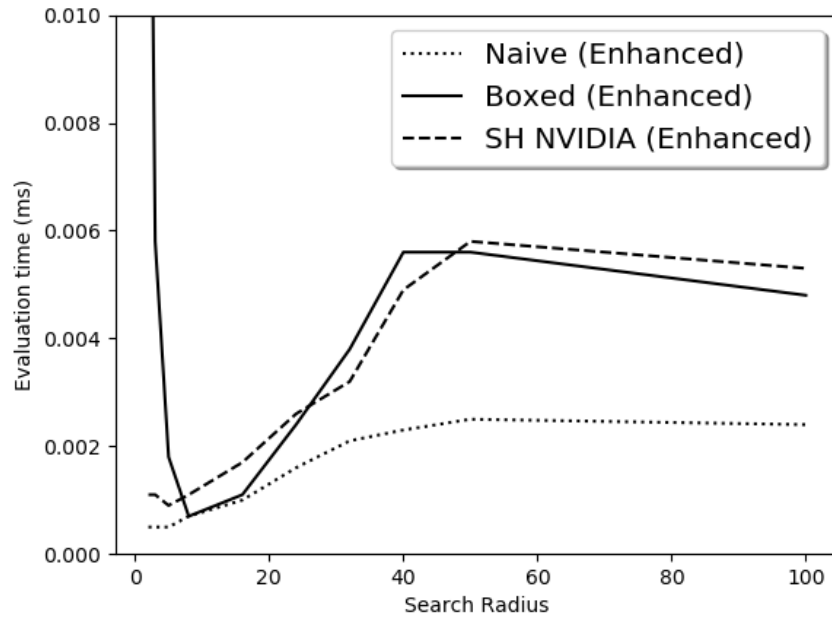


Рисунок А.37 – Порівняння часу обчислення наївного, боксового та алгоритму просторового хешування (хеш-функція NVIDIA) із застосуванням шаблонів та багатонитковості для $r = 2, 3, 5, 8, 16, 24, 32, 40, 50, 100$

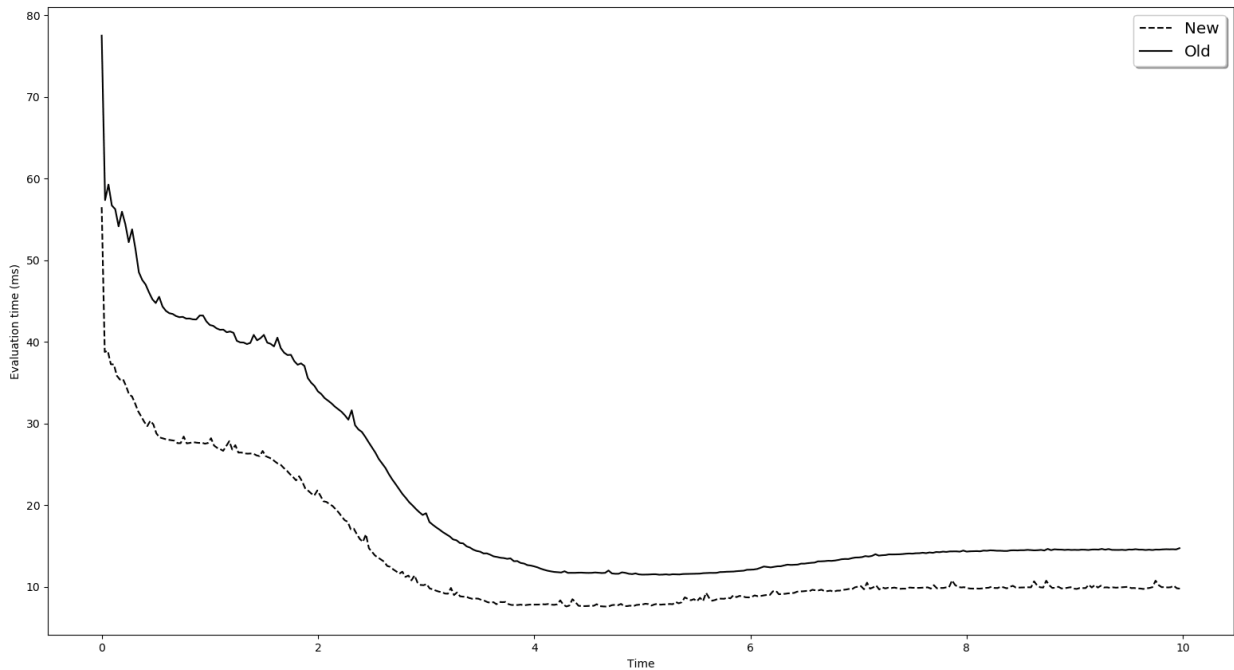


Рисунок А.38 – Порівняння часу обчислень для оптимізованої та неоптимізованої симуляції течії рідини

ДОДАТОК Б

Вихідний код програми

NNS/Common.h

```

1  #pragma once
2
3  #include <set>
4  #include <array>
5  #include <cmath>
6  #include <mutex>
7  #include <vector>
8  #include <memory>
9  #include <random>
10 #include <thread>
11 #include <iostream>
12 #include <unordered_map>
13
14
15 namespace SPHAlgorithms::NNS {
16
17     // Compilation control
18     #define SPH_OPTIMIZE_TEMPLATES
19     // #define SPH_OPTIMIZE_MULTITHREADING
20     namespace Algorithm {
21         enum {
22             Naive, // No assumption at all
23             Boxed, // Classic Algorithm
24             #if !defined(SPH_OPTIMIZE_TEMPLATES) && !defined(SPH_OPTIMIZE_MULTITHREADING)
25                 Boxed_Mod0, // Count all central points as neighbors
26                 Boxed_Mod1, // Count all as neighbors
27                 Boxed_Mod2, // Lower box radius, so all internal points are neighbors
28                 Boxed_Mod3, // Lower box radius, count all as neighbors
29                 SH, // Classic spatial hash Algorithm
30             #endif
31             SH_Nvidia, // SP with nvidia hash
32             #if !defined(SPH_OPTIMIZE_TEMPLATES) && !defined(SPH_OPTIMIZE_MULTITHREADING)
33                 SH_Mod0, // Count all central points as neighbors
34                 SH_Mod1, // Count all as neighbors
35             #endif
36             Max
37         };
38         static std::string Text[Algorithm::Max] = {
39             "Naive",
40             "Boxed",
41             #if !defined(SPH_OPTIMIZE_TEMPLATES) && !defined(SPH_OPTIMIZE_MULTITHREADING)
42                 "Boxed_Mod0",
43                 "Boxed_Mod1",
44                 "Boxed_Mod2",
45                 "Boxed_Mod3",
46                 "SH",
47             #endif
48             "SH Nvidia",
49             #if !defined(SPH_OPTIMIZE_TEMPLATES) && !defined(SPH_OPTIMIZE_MULTITHREADING)
50                 "SH_Mod0",
51                 "SH_Mod1"
52             #endif
53         };
54     };
55
56
57
58
59     extern size_t s_num_threads;
60     size_t s_num_threads = std::thread::hardware_concurrency();

```

```

61
62 template<typename _T, size_t _Dim, typename = std::make_index_sequence<_Dim>>
63 class Point;
64
65 template<typename _T, size_t _Dim, std::size_t ... _Idx>
66 class Point<_T, _Dim, std::index_sequence<_Idx ...>> : public std::array<_T, _Dim>
67 {
68     public:
69         bool within(const Point<_T, _Dim> & _other, const _T & _range) const
70         {
71             #ifdef SPH_OPTIMIZE_TEMPLATES
72                 _T accumulated = ( std::pow(_other[_Idx] - (*this)[_Idx], 2) + ... );
73                 if ( accumulated > _range * _range )
74                     return false;
75             #else
76                 _T accumulated = 0;
77                 for ( size_t i = 0; i < _Dim; ++i ) {
78                     accumulated += std::pow(_other[i] - (*this)[i], 2);
79                     if ( accumulated > _range * _range )
80                         return false;
81                 }
82             #endif
83             return true;
84         }
85         bool operator==(const Point<_T, _Dim,
86                         std::index_sequence<_Idx ...>& _other) const
87         {
88             bool res = (true & ... & ((*this)[_Idx] == _other[_Idx]) );
89             return res;
90         }
91     };
92
93
94
95
96 template<typename _T, size_t _Dim>
97 class Boundary
98 {
99     public:
100         Boundary() = delete;
101         Boundary(const Boundary<_T, _Dim> & _other) = default;
102         Boundary(Boundary<_T, _Dim> && _other) = default;
103         ~Boundary() = default;
104         Boundary & operator=(const Boundary &) noexcept = default;
105         Boundary & operator=(Boundary &&) noexcept = delete;
106
107         std::pair< Point<_T, _Dim>, Point<_T, _Dim> > pair;
108     };
109
110
111 template<typename _T, size_t _Dim>
112 class DataEntry
113 {
114     public:
115         Point<_T, _Dim> point;
116         std::vector<const DataEntry<_T, _Dim> *> neighbors;
117
118         explicit DataEntry(const DataEntry & _other) = default;
119         explicit DataEntry(DataEntry && other) = default;
120

```

```

121
122     DataEntry(const Point<_T, _Dim> & _point) = delete;
123     explicit DataEntry() = delete;
124     DataEntry & operator=(const DataEntry &) noexcept = delete;
125     DataEntry & operator=(DataEntry &&) noexcept = delete;
126
127     DataEntry(Point<_T, _Dim> && _point) :
128         point{ std::forward< Point<_T, _Dim> >(_point) }
129     {}
130 };
131
132
133 enum BoxedFlags
134 {
135     CentralNeighborAcceptance = 0x1,
136     OuterNeighborAcceptance = 0x2,
137     IncircledRange = 0x4,
138     HashFunctionNvidia = 0x8
139 };
140
141 template<typename _T>
142 constexpr _T get_range(_T _range, uint8_t _flags)
143 {
144     if ( _flags & IncircledRange )
145         return _range / std::sqrt(2);
146     else
147         return _range;
148 }
149
150
151 /*
152  * This class represents a 3D relative box position.
153  * @x, @y, @z = { -1, 0, 1}, so there are total 27 possible relative positions.
154  * TBD fold expression
155  */
156 template<size_t _Dim>
157 class BoxRelativePosition
158 {
159 public:
160     std::array<std::int8_t, _Dim> pos;
161
162     /*
163     * Returns opposite relation for box. So if one box has relative position
164     * {-1, 0, -1} regarding the second box then the second box has relative
165     * position {1, 0, 1} regarding the first box.
166     */
167     constexpr BoxRelativePosition<_Dim> opposite() const
168     {
169         return BoxRelativePosition<_Dim>
170         {
171             static_cast<std::int8_t>(~pos[0]),
172             static_cast<std::int8_t>(~pos[1]),
173             static_cast<std::int8_t>(~pos[2])
174         };
175     }
176
177     /*
178     * x, y, z belong to [-1,1]
179     * So if we inc them by one, they would belong to [0..2] (3 elements)
180

```

```

181      * So the array number is (x+1)*3*3 + (y+1)*3 + (z+1);
182      * max range is (2*9 + 2*3 + 2) = 26 -- resulting 27 elements
183      *
184      * @return 0..26
185      */
186      inline size_t to_array_index() const
187      {
188          return static_cast<uint8_t>(pos[0] + 1)*3*3 +
189              static_cast<uint8_t>(pos[1] + 1)*3 +
190              static_cast<uint8_t>(pos[2] + 1);
191      }
192  };
193
194  /*
195   * TBD fold expression
196   */
197  template<typename _T, size_t _Dim>
198  const std::array< BoxRelativePosition<_Dim>, _Dim * _Dim * _Dim> box_positions =
199  {
200      {-1, -1, -1},
201      {-1, -1, 0},
202      {-1, -1, +1},
203
204      {-1, 0, -1},
205      {-1, 0, 0},
206      {-1, 0, +1},
207
208      {-1, +1, -1},
209      {-1, +1, 0},
210      {-1, +1, +1},
211
212
213      { 0, -1, -1},
214      { 0, -1, 0},
215      { 0, -1, +1},
216
217      { 0, 0, -1},
218      { 0, 0, 0},
219      { 0, 0, +1},
220
221      { 0, +1, -1},
222      { 0, +1, 0},
223      { 0, +1, +1},
224
225
226      {+1, -1, -1},
227      {+1, -1, 0},
228      {+1, -1, +1},
229
230      {+1, 0, -1},
231      {+1, 0, 0},
232      {+1, 0, +1},
233
234      {+1, +1, -1},
235      {+1, +1, 0},
236      {+1, +1, +1},
237  };
238
239  }
240

```

NNS/Naive.h

```

1  #pragma once
2
3
4  #include "NNS/Common.h"
5
6  namespace SPHAlgorithms::NNS {
7
8      template<typename _T, size_t _Dim>
9      static void Naive_worker(std::vector< NNS::DataEntry<_T, _Dim> > & _data,
10                               const _T _range,
11                               size_t _i)
12      {
13          auto it = _data.begin();
14          it += _i;
15          for ( ; it < _data.end(); it += s_num_threads) {
16              for ( auto jt = _data.begin(); jt != _data.end(); jt ++ ) {
17                  if ( jt == it )
18                      continue;
19                  if ( it->point.within(jt->point, _range) )
20                      it->neighbors.push_back( &(*jt) );
21              }
22          }
23      }
24
25      /*
26       * Naive algorithm of finding nearest neighbors for each of @data
27       * containers.
28       *
29       * @data      -- Const reference to a vector of shared data. Datas are going
30       * to be filled in. Won't be clean up in this method.
31       * @boundary  -- Boundaries in which points are distributed. It is used for
32       * more effective work.
33       * @range     -- Maximum range between neighbors.
34       */
35
36      template<typename _T, size_t _Dim>
37      void Naive(std::vector< NNS::DataEntry<_T, _Dim> > & _data,
38                const _T _range)
39      {
40          if ( !_data.size() ) {
41              std::cout << "Empty data input" << std::endl;
42              return;
43          }
44          #ifdef SPH_OPTIMIZE_MULTITHREADING
45              std::vector<std::thread> pool;
46              pool.reserve( s_num_threads );
47              for ( size_t i = 0; i < s_num_threads; i++ )
48                  pool.emplace_back( std::thread(Naive_worker<_T, _Dim>,
49                                                  std::ref(_data), _range, i) );
50              for ( auto & it : pool )
51                  it.join();
52          #else
53              for ( auto it = _data.begin(); it != _data.end() - 1; ++it )
54                  for ( auto jt = it + 1; jt != _data.end(); ++jt )
55                      if ( it->point.within(jt->point, _range) ) {
56                          it->neighbors.push_back( &(*jt) );
57                          jt->neighbors.push_back( &(*it) );
58                      }
59          #endif
60      }
61
62  }
63

```

NNS/Boxed.h

```

1  #pragma once
2
3  #include "NNS/Common.h"
4
5  namespace SPHAlgorithms::NNS {
6
7
8      /*
9      * Class represents a box.
10     * @x -- Integer X position value
11     * @y -- Integer Y position value
12     * @z -- Integer Z position value
13     *
14     * @fNotProcessed -- Bool flag that shows if this box has already been
15     * processed or not.
16     *
17     * @data -- Vector of data pointers, whose .point field gets into this box.
18     *
19     */
20     template<typename T, size_t _Dim>
21     class Box
22     {
23     public:
24         Box(size_t _size)
25         {
26             inner_points.reserve(_size);
27         }
28         std::array<long, _Dim> pos;
29
30         /*
31         * Mask that contains info about visited boxes
32         */
33         bool visited[_Dim * _Dim * _Dim] = { false };
34
35         /*
36         * This function
37         * @x, @y, @z -- one of the values {-1, 0, 1}
38         *
39         * @return :
40         *     1) true -- visit is required
41         *     2) false -- visit has been already done
42         *
43         */
44         inline bool visit(const BoxRelativePosition<_Dim> & _pos);
45
46         std::vector< DataEntry<T, _Dim> * > inner_points;
47     };
48
49     /*
50     * TBD fold expressions
51     */
52     template<typename _T, size_t _Dim>
53     class Boxes
54     {
55     {
56
57     public:
58
59         /*
60         * @x, @y, @z -- Maximum possible axis values.

```

```

61         * @size -- Total elements, memory will not be spared to avoid
62         * reallocation.
63         */
64         explicit Boxes(long _x, long _y, long _z, size_t _size);
65
66         /*
67         * Returns box at @x, @y, @z position.
68         * Return nullptr if there's not such box.
69         */
70         inline Box<T, _Dim> * at(long _x, long _y, long _z);
71
72         /*
73         * Adds a @data pointer to box at @x, @y, @z.
74         * If box does not exist, does nothing.
75         */
76         inline void add(long _x, long _y, long _z,
77                        DataEntry<T, _Dim> & _data);
78
79         /*
80         * Returns vector of boxes, that are neighbors within 3D space of
81         * box at @x, @y, @z and are NOT marked as visited.
82         * Marks returned boxes as visited.
83         */
84         std::vector< Box<T, _Dim> * > neighborsOf(long _x, long _y, long _z);
85
86         std::vector< Box<T, _Dim> > boxes;
87
88     private:
89         const long m_x, m_y, m_z;
90         const size_t m_size;
91
92 };
93
94
95 template<typename T, size_t _Dim>
96 inline bool Box<T, _Dim>::visit(const BoxRelativePosition<_Dim> & _pos)
97 {
98     size_t idx = _pos.to_array_index();
99     if ( idx > _Dim * _Dim * _Dim - 1 ) {
100         std::cout << "Box::visit unexpected index " << idx << std::endl;
101         return false;
102     }
103     if ( visited[idx] )
104         return false;
105     visited[idx] = true;
106
107     return true;
108 }
109
110 template<typename T, size_t _Dim>
111 Boxes<T, _Dim>::Boxes(long _x, long _y, long _z, size_t _size) :
112     boxes( ( _x + 1 ) * ( _y + 1 ) * ( _z + 1 ) + 1, ( _x + 1 ) * ( _y + 1 ) * ( _z + 1 ) + 1 ),
113     m_x(_x + 1), m_y(_y + 1), m_z(_z + 1), m_size(_size)
114 {
115     for ( auto & it : boxes )
116         it = Box<T, _Dim>(_size);
117 }
118
119 template<typename T, size_t _Dim>
120 Box<T, _Dim> * Boxes<T, _Dim>::at(long x, long y, long z)

```

```

121 {
122     if ( _x < 0 || _y < 0 || _z < 0 ||
123         _x >= m_x || _y >= m_y || _z >= m_z )
124     {
125         return nullptr;
126     }
127     return &boxes[_x * m_y * m_z + _y * m_z + _z];
128 }
129
130 template<typename _T, size_t _Dim>
131 inline void Boxes<_T, _Dim>::add(long _x, long _y, long _z,
132                                   DataEntry<_T, _Dim> & _data)
133 {
134     if ( _x >= m_x || _y >= m_y || _z >= m_z ) {
135         std::cout << "Trying to add out-of-boundary point" << std::endl;
136         return;
137     }
138     Box<_T, _Dim> * box = nullptr;
139     if ( (box = this->at(_x, _y, _z)) == nullptr ) {
140         std::cout << "Warning: protected from adding a non-valid box ("
141                     << _x << ", " << _y << ", " << _z << ")" << std::endl;
142         return;
143     }
144     box->inner_points.push_back(&_data);
145 }
146
147
148
149
150 template<typename _T, size_t _Dim>
151 std::vector< Box<_T, _Dim> * > Boxes<_T, _Dim>::neighborsOf(long _x, long _y, long _z)
152 {
153     std::vector< Box<_T, _Dim> * > neighbors;
154     neighbors.reserve(m_size);
155     Box<_T, _Dim> * box;
156     Box<_T, _Dim> * this_box;
157     if ( !(this_box = this->at(_x, _y, _z)) ) {
158         std::cout << "neighborsOf: Bad 'this' box "
159                     << _x << " " << _y << " " << _z << std::endl;
160         return neighbors;
161     }
162     /*
163     * Try visit this_box's each of the 3x3 cube bound
164     * subcubes (and visit that box's opposite subcube,
165     * which this_cube actually is.
166     */
167     for ( const auto & it : box_positions<_T, _Dim> )
168         if ( (box = this->at(_x + it.pos[0],
169                             _y + it.pos[1],
170                             _z + it.pos[2]))
171 #ifndef SPH_OPTIMIZE_MULTITHREADING
172             && this_box->visit(it)
173             && box->visit( it.opposite() )
174 #endif
175         )
176         {
177             neighbors.push_back(box);
178         }
179     return neighbors;
180 }

```

```

181     }
182
183
184     template<typename _T, size_t _Dim>
185     constexpr void boxed_process_central(Box<_T, _Dim> * _box, _T _range, uint8_t _flags)
186     {
187         bool f = (_flags & (CentralNeighborAcceptance | OuterNeighborAcceptance)) > 0;
188         for ( auto it = _box->inner_points.begin(); it != _box->inner_points.end() - 1; ++it )
189             /* For every particle pair within the box */
190             for ( auto jt = it + 1; jt != _box->inner_points.end(); ++jt )
191                 if ( f || (*jt)->point.within( (*it)->point, _range) ) {
192                     (*it)->neighbors.push_back( *jt );
193                     (*jt)->neighbors.push_back( *it );
194                 }
195     }
196
197     template<typename _T, size_t _Dim>
198     constexpr void boxed_process_outer(std::vector< Box<_T, _Dim> * > & _neighbors,
199                                     Box<_T, _Dim> * _box, _T _range, uint8_t _flags)
200     {
201         bool f = (_flags & OuterNeighborAcceptance) > 0;
202         // for each box within neighbors set
203         for ( const auto & it : _neighbors )
204             // for each inner_points object within that neighbor
205             for ( const auto & jt : it->inner_points )
206                 // and each inner_points object within current box
207                 for ( const auto & kt : _box->inner_points )
208                     if ( f || jt->point.within( kt->point, _range) ) {
209                         kt->neighbors.push_back( jt );
210                         jt->neighbors.push_back( kt );
211                     }
212     }
213
214
215     template<typename _T, size_t _Dim> constexpr
216     void boxed_process_outer_atomic(std::vector< Box<_T, _Dim> * > & _neighbors,
217                                   Box<_T, _Dim> * _box, _T _range, uint8_t _flags)
218     {
219         bool f = (_flags & OuterNeighborAcceptance) > 0;
220         // for each box within neighbors set
221         for ( const auto & it : _neighbors )
222             // for each inner_points object within that neighbor
223             for ( const auto & jt : it->inner_points )
224                 // and each inner_points object within current box
225                 for ( const auto & kt : _box->inner_points )
226                     if ( f || jt->point.within( kt->point, _range) )
227                         kt->neighbors.push_back( jt );
228     }
229
230     template<typename _T, size_t _Dim>
231     static void Boxed_worker(Boxes<_T, _Dim> & _boxes,
232                             const Boundary<_T, 3> & _boundary,
233                             _T _range,
234                             size_t _i,
235                             uint8_t _flags = 0)
236     {
237         const _T range = get_range( _range, _flags );
238         long x_b = static_cast<long>( ( _boundary.pair.second[0] - _boundary.pair.first[0] ) / range );
239         long y_b = static_cast<long>( ( _boundary.pair.second[1] - _boundary.pair.first[1] ) / range );
240         long z_b = static_cast<long>( ( _boundary.pair.second[2] - _boundary.pair.first[2] ) / range );

```

```

241     size_t counter = 0;
242
243     for ( long i = 0; i <= x_b; ++ i ) {
244         for ( long j = 0; j <= y_b; ++ j ) {
245             for ( long k = 0; k <= z_b; ++ k ) { /* For each box within the cube */
246                 if ( counter % s_num_threads != _i ) {
247                     counter ++;
248                     continue;
249                 }
250                 counter ++;
251
252                 Box<_T, _Dim> * box = _boxes.at(i, j, k);
253                 if ( !box || box->inner_points.size() == 0 )
254                     continue;
255                 boxed_process_central(box, range, _flags);
256                 std::vector< Box<_T, _Dim> * > neighbors = _boxes.neighborsOf(i, j, k);
257                 boxed_process_outer_atomic(neighbors, box, range, _flags);
258             } } }
259     }
260
261     template<typename _T, size_t _Dim>
262     void Boxed(std::vector< NNS::DataEntry<_T, 3> > & _data,
263               const Boundary<_T, 3> & _boundary,
264               _T _range,
265               uint8_t _flags = 0)
266     {
267         const _T range = get_range(_range, _flags);
268         // Add box indexes
269         long x_b = static_cast<long>( (_boundary.pair.second[0] - _boundary.pair.first[0]) / range );
270         long y_b = static_cast<long>( (_boundary.pair.second[1] - _boundary.pair.first[1]) / range );
271         long z_b = static_cast<long>( (_boundary.pair.second[2] - _boundary.pair.first[2]) / range );
272
273         Boxes<_T, _Dim> boxes( x_b, y_b, z_b, _data.size() );
274
275         // Add box indexes & fill boxes struct
276         for ( auto & it : _data )
277             boxes.add(
278                 static_cast<long>( (it.point[0] - _boundary.pair.first[0]) / range ),
279                 static_cast<long>( (it.point[1] - _boundary.pair.first[1]) / range ),
280                 static_cast<long>( (it.point[2] - _boundary.pair.first[2]) / range ),
281                 it
282             );
283     #ifdef SPH_OPTIMIZE_MULTITHREADING
284         std::vector<std::thread> pool;
285         pool.reserve( s_num_threads );
286         for ( size_t i = 0; i < s_num_threads; i++ ) {
287             pool.emplace_back( std::thread(Boxed_worker<_T, _Dim>, std::ref(boxes),
288                                           std::ref(_boundary), _range, i, _flags) );
289         }
290         for ( auto & it : pool )
291             it.join();
292     #else
293         for ( long i = 0; i <= x_b; ++ i ) {
294             for ( long j = 0; j <= y_b; ++ j ) {
295                 for ( long k = 0; k <= z_b; ++ k ) { /* For each box within the cube */
296
297                     Box<_T, _Dim> * box = boxes.at(i, j, k);
298                     if ( !box || box->inner_points.size() == 0 )
299                         continue;
300                     boxed_process_central(box, range, _flags);
301
302                     std::vector< Box<_T, _Dim> * > neighbors = boxes.neighborsOf(i, j, k);
303                     boxed_process_outer(neighbors, box, range, _flags);
304                 } } }
305             }
306         }
307     }
308 
```

NNS/SpatialHash.h

```

1  #pragma once
2
3
4  #include "NNS/Boxed.h"
5
6  namespace SPHAlgorithms::NNS {
7
8      /* Spatial Box Struct,
9       * holds just a vector of inner points, nothing special
10      */
11      template<typename _T, size_t _Dim>
12      class SHBox
13      {
14      public:
15          SHBox(DataEntry<_T, _Dim> * _entry, size_t _size)
16          {
17              inner_points.reserve(_size);
18              inner_points.push_back(_entry);
19          }
20          ~SHBox() = default;
21
22          std::vector< DataEntry<_T, _Dim> *> inner_points;
23          std::set<size_t> visited_hash;
24
25      };
26
27      namespace SpatialHash
28      {
29          constexpr size_t mid(size_t low, size_t high) {
30              return (low + high) / 2;
31          }
32
33          // precondition: low*low <= n, high*high > n.
34          constexpr size_t ceilsqrt(size_t n, size_t low, size_t high) {
35              return low + 1 >= high
36                  ? high
37                  : (mid(low, high) * mid(low, high) == n)
38                      ? mid(low, high)
39                      : (mid(low, high) * mid(low, high) < n)
36                          ? ceilsqrt(n, mid(low, high), high)
37                          : ceilsqrt(n, low, mid(low, high));
42          }
43
44          // returns ceiling(sqrt(n))
45          constexpr size_t ceilsqrt(size_t n) {
46              return n < 3
47                  ? n
48                  : ceilsqrt(n, 1, size_t(1) << (std::numeric_limits<size_t>::digits / 2));
49          }
50
51          // returns true if n is divisible by an odd integer in
52          // [2 * low + 1, 2 * high + 1).
53          constexpr bool find_factor(size_t n, size_t low, size_t high)
54          {
55              return low + 1 >= high
56                  ? (n % (2 * low + 1)) == 0
57                  : (find_factor(n, low, mid(low, high))
58                      || find_factor(n, mid(low, high), high));
59          }
60      }

```

```

61
62
63     template<size_t _N>
64     constexpr size_t large_enough_prime()
65     {
66         if ( _N < 11 ) // BS guard
67             return 11;
68         size_t n = _N;
69         do {
70             if ( n % 2 == 0 ) {
71                 n++;
72                 continue;
73             }
74             if ( !find_factor(n, 1, (ceil(sqrt(n)) + 1) / 2) ) {
75                 return n;
76             }
77             n += 2;
78         } while ( true );
79     }
80
81     constexpr size_t p1()
82     {
83         return 73856093;
84     }
85     constexpr size_t p2()
86     {
87         return 19349663;
88     }
89     constexpr size_t p3()
90     {
91         return 83492791;
92     }
93 }
94
95
96
97     template<typename _T, size_t _Dim>
98     constexpr void sh_process_central(const SHBox<_T, _Dim> * _box, _T _range, uint8_t _flags)
99     {
100         bool f = true & ( (_flags & (CentralNeighborAcceptance | OuterNeighborAcceptance)) > 0 );
101         for ( auto it = _box->inner_points.begin(); it != _box->inner_points.end() - 1; ++it )
102             /* For every particle pair within the box */
103             for ( auto jt = it + 1; jt != _box->inner_points.end(); ++jt )
104                 if ( f || (*jt)->point.within( (*it)->point, _range) ) {
105                     (*it)->neighbors.push_back( *jt );
106                     (*jt)->neighbors.push_back( *it );
107                 }
108     }
109
110     template<typename _T, size_t _Dim>
111     constexpr void sh_process_outer(std::vector< DataEntry<_T, _Dim> * > & _neighbors,
112                                     DataEntry<_T, _Dim> * _point, _T _range, uint8_t _flags)
113     {
114         bool f = true & ( (_flags & (OuterNeighborAcceptance)) > 0 );
115         for ( const auto & it : _neighbors ) {
116             if ( f || _point->point.within(it->point, _range) ) {
117                 _point->neighbors.push_back( &(*it) );
118                 it->neighbors.push_back( &(*_point) );
119             }
120         }

```

```

121     }
122
123     template<typename _T, size_t _Dim>
124     constexpr void sh_process_outer_atomic(std::vector< DataEntry<_T, _Dim> * > & _neighbors,
125                                           DataEntry<_T, _Dim> * _point,
126                                           _T _range, uint8_t _flags)
127     {
128         bool f = true & ((_flags & (OuterNeighborAcceptance)) > 0);
129         for ( const auto & it : _neighbors ) {
130             if ( f || _point->point.within(it->point, _range) )
131                 _point->neighbors.push_back( &(*it) );
132         }
133     }
134
135     // Works only for 3D
136     template<typename _T>
137     constexpr size_t calculate_hash_kelager(const Point<_T, 3> & _entry, _T _range, size_t _n)
138     {
139         return ( ( static_cast<size_t>(_entry[0] / _range) * SpatialHash::p1() ) ^
140                 ( static_cast<size_t>(_entry[1] / _range) * SpatialHash::p2() ) ^
141                 ( static_cast<size_t>(_entry[2] / _range) * SpatialHash::p3() ) ) % _n;
142     }
143
144     template<typename _T, size_t _Dim, size_t ... _Idx>
145     static size_t _M_calculate_hash_nvidia_helper(const Point<_T, _Dim> & _entry, _T _range,
146                                                  std::index_sequence<_Idx ...>)
147     {
148         /*
149          * long is 32 bit, lets fit it
150          * XXXX
151          *   XXXX
152          *   XXXX .. etc
153          * This way the part will always take 32 bit, so the space we will divide
154          * will be equal to bits(size_t) - bits(long) - 1. Then we will divide it
155          * to _Dim - 1 and get the desired shift, multiplying it to _Idx we are
156          * getting proper shift for fold expressions
157          */
158         return (( (static_cast<long>(_entry[_Idx] / _range) <<
159                    (( 8*(sizeof(size_t) - sizeof(long)) - 1) / (_Dim - 1) ) * _Idx ) + ... ));
160     }
161
162     template<typename _T, size_t _Dim>
163     constexpr size_t calculate_hash_nvidia(const Point<_T, _Dim> & _entry, _T _range)
164     {
165         return _M_calculate_hash_nvidia_helper(_entry, _range,
166                                                std::make_index_sequence<_Dim>() );
167     }
168
169     template<typename _T, size_t _Dim>
170     constexpr size_t calculate_hash(const Point<_T, _Dim> & _entry,
171                                    size_t _n, _T _range, uint8_t _flags)
172     {
173         if ( _flags & HashFunctionNvidia )
174             return calculate_hash_nvidia<_T>(_entry, _range);
175         static_assert ( _Dim == 3, "Kelager function is defined for 3D space only");
176         return calculate_hash_kelager<_T>(_entry, _range, _n);
177     }
178
179
180     template<typename T, size_t Dim, size_t N>

```

```

181 static void SH_worker(std::unordered_map< size_t, SHBox<_T, _Dim> > & _hash_boxes,
182                      _T _range,
183                      size_t _i,
184                      uint8_t _flags = 0)
185 {
186     const size_t n = SpatialHash::large_enough_prime<N>();
187     size_t counter = 0;
188
189     for ( const auto & it : _hash_boxes ) {
190         if ( counter % s_num_threads != _i ) {
191             counter ++;
192             continue;
193         }
194         counter ++;
195
196         sh_process_central<_T>(&it.second, _range, _flags);
197
198         for ( const auto & jt : it.second.inner_points ) {
199             std::set<size_t> hashes_to_visit;
200             for ( const auto & pos : box_positions<_T, _Dim> ) {
201                 size_t hash = calculate_hash<_T, _Dim>(Point<_T, _Dim>{
202                                     // TBD fold expression
203                                     jt->point[0] + _range * pos.pos[0],
204                                     jt->point[1] + _range * pos.pos[1],
205                                     jt->point[2] + _range * pos.pos[2]
206                                     }, n, _range, _flags );
207                 if ( it.first == hash || !_hash_boxes.count(hash) )
208                     continue;
209                 hashes_to_visit.insert(hash);
210             }
211             std::vector< DataEntry<_T, _Dim> * > possible_neighbors;
212             for ( const auto & kt : hashes_to_visit ) {
213                 SHBox<_T, _Dim> & box = _hash_boxes.at(kt);
214                 possible_neighbors.insert( possible_neighbors.end(),
215                                           box.inner_points.begin(),
216                                           box.inner_points.end() );
217             }
218             sh_process_outer_atomic(possible_neighbors, jt, _range, _flags);
219         }
220     }
221 }
222
223
224
225 template<typename _T, size_t _Dim, size_t _N>
226 void SH(std::vector< NNS::DataEntry<_T, _Dim> > & _data,
227         _T _range,
228         uint8_t _flags = 0)
229 {
230     const _T range = get_range(_range, _flags);
231     const size_t n = SpatialHash::large_enough_prime<N>();
232
233     std::unordered_map< size_t, SHBox<_T, _Dim> > hash_boxes;
234     hash_boxes.reserve(_data.size() * 2);
235
236     // Add box indexes & fill boxes struct
237     for ( auto & it : _data ) {
238         auto hs = calculate_hash<_T, _Dim>(it.point, n, range, _flags);
239         if ( hash_boxes.count(hs) )
240             hash_boxes.at(hs).inner_points.push back(&it);

```

```

241         else
242             hash_boxes.emplace( hs, SHBox<T, _Dim>(&it, _data.size()) );
243         }
244
245 #ifdef SPH_OPTIMIZE_MULTITHREADING
246     std::vector<std::thread> pool;
247     pool.reserve( s_num_threads );
248     for ( size_t i = 0; i < s_num_threads; i++ ) {
249         pool.emplace_back( std::thread(SH_worker<T, _Dim, _N>,
250                                     std::ref(hash_boxes), range, i, _flags) );
251     }
252     for ( auto & it : pool )
253         it.join();
254 #else
255     for ( const auto & it : hash_boxes ) {
256         sh_process_central<T>(&it.second, range, _flags);
257
258         for ( const auto & jt : it.second.inner_points ) {
259             std::set<size_t> hashes_to_visit;
260             for ( const auto & pos : box_positions<T, _Dim> ) {
261                 size_t hash = calculate_hash<T, _Dim>(Point<T, _Dim>{
262                                     // TBD fold expression
263                                     jt->point[0] + range * pos.pos[0],
264                                     jt->point[1] + range * pos.pos[1],
265                                     jt->point[2] + range * pos.pos[2]
266                                     }, n, range, _flags );
267
268                 if ( it.first == hash ||
269                     !hash_boxes.count(hash) ||
270                     it.second.visited_hash.count(hash) )
271                 {
272                     continue;
273                 }
274                 hashes_to_visit.insert(hash);
275             }
276             std::vector< DataEntry<T, _Dim> * > possible_neighbors;
277             for ( const auto & kt : hashes_to_visit ) {
278                 SHBox<T, _Dim> & box = hash_boxes.at(kt);
279                 box.visited_hash.insert(it.first);
280                 possible_neighbors.insert( possible_neighbors.end(),
281                                     box.inner_points.begin(),
282                                     box.inner_points.end() );
283             }
284             sh_process_outer(possible_neighbors, jt, range, _flags);
285         }
286     }
287 }
288 #endif
289
290 }
291
292 }
293

```

main.cpp

```

1  #include "NNS/Naive.h"
2  #include "NNS/Boxed.h"
3  #include "NNS/SpatialHash.h"
4
5  #include <map>
6  #include <chrono>
7
8
9  namespace SPHAlgorithms::NNS {
10
11     template<typename _T, size_t _Dim, size_t ... _Idx>
12     static void _M_GenerateHelper(std::vector< NNS::DataEntry<_T, _Dim> > & _cont,
13                                   size_t _count,
14                                   const Boundary<_T, _Dim> & _boundary,
15                                   std::index_sequence<_Idx ...>)
16     {
17         std::random_device r;
18         std::default_random_engine re( r() );
19         std::uniform_real_distribution<_T> unif[_Dim];
20         _cont.clear();
21         _cont.reserve(_count);
22
23         (( unif[_Idx] = std::uniform_real_distribution<_T>(_boundary.pair.first[_Idx],
24                                                           _boundary.pair.second[_Idx]) ), ...);
25
26         for ( size_t i = 0; i < _count; ++ i)
27             _cont.emplace_back( Point<_T, _Dim>{ { ( unif[_Idx](re) )... } } );
28     }
29
30     /*
31     * This function generates random particles and adds them to container
32     *
33     * @cont      -- Container with shared Data pointers
34     * @number    -- Number of point to generate
35     * @boundary  -- Specified range of points
36     *
37     */
38     template<typename _T, size_t _Dim>
39     [[maybe_unused]] static void Generate(std::vector< NNS::DataEntry<_T, _Dim> > & _cont,
40                                           size_t _count,
41                                           const Boundary<_T, _Dim> & _boundary)
42     {
43         _M_GenerateHelper<_T, _Dim>(_cont, _count, _boundary, std::make_index_sequence<_Dim>());
44     }
45
46
47     template <typename _T, size_t _Dim, size_t _N>
48     class NNSInstance
49     {
50     public:
51         const NNS::Boundary<_T, _Dim> boundary;
52         std::vector< NNS::DataEntry<_T, 3> > points;
53
54
55         NNSInstance(const NNS::Boundary<_T, 3> & _boundary,
56                    const std::vector< NNS::DataEntry<_T, 3> > & _points) :
57             boundary(_boundary),
58             points(_points)
59         {}
60

```



```

121 |                                     IncircledRange);
122 |     case Algorithm::SH:
123 |         return NNS::SH<double, 3, _N>(points, range);
124 | #endif
125 |     case Algorithm::SH_Nvidia:
126 |         return NNS::SH<double, 3, _N>(points, range, HashFunctionNvidia);
127 | #if !defined(SPH_OPTIMIZE_TEMPLATES) && !defined(SPH_OPTIMIZE_MULTITHREADING)
128 |     case Algorithm::SH_Mod0:
129 |         return NNS::SH<double, 3, _N>(points, range, CentralNeighborAcceptance);
130 |     case Algorithm::SH_Mod1:
131 |         return NNS::SH<double, 3, _N>(points, range, CentralNeighborAcceptance |
132 |                                     OuterNeighborAcceptance);
133 | #endif
134 |
135 |     }
136 | }
137 |
138 |
139 | };
140 |
141 |
142 | template<typename _T, size_t _Dim, size_t _N>
143 | class TestNNS
144 | {
145 | public:
146 |     const NNS::Boundary<_T, _Dim> boundary;
147 |
148 |     TestNNS(NNS::Boundary<_T, _Dim> && _boundary) :
149 |         boundary { std::forward<NNS::Boundary<_T, _Dim>>(_boundary) }
150 |     {}
151 |
152 |     /*
153 |     * This function launches specified test on a specified set of points
154 |     * and returns <evaluation time, accuracy> pair
155 |     *
156 |     */
157 |     std::pair<double, double> launch_test(std::vector< NNS::DataEntry<_T, _Dim> > & _points,
158 |                                         _T _range,
159 |                                         int _algorithm,
160 |                                         const NNSInstance<_T, _Dim, _N> & _perfect)
161 |     {
162 |         NNSInstance<_T, _Dim, _N> naive(boundary, _points);
163 |         auto start = std::chrono::high_resolution_clock::now();
164 |         naive.run(_algorithm, _range);
165 |         auto elapsed_seconds = std::chrono::high_resolution_clock::now() - start;
166 |         return std::make_pair( elapsed_seconds.count(), naive.compare(_perfect) );
167 |     }
168 |
169 |
170 |     void test()
171 |     {
172 |         std::cout << "Running for " << _N << " particles" << std::endl;
173 |         std::array<_T, 10> range_list = {2., 3., 5., 8., 16., 24., 32., 40., 50., 100.};
174 |         // For each range it contains a map of algorithm results
175 |         // (vector of 10 tests with pair of results)
176 |         std::vector< std::vector< std::map<int, std::pair<double, double> > > > results(10);
177 |
178 |         for ( size_t i = 0; i < range_list.size(); ++i ) {
179 |             auto & vector_ref = results[ i ];
180 |             for ( int j = 0; j < 50; ++j ) {

```

```

181         std::map<int, std::pair<double, double> > test_map;
182
183         std::vector< NNS::DataEntry<_T, _Dim> > test_points;
184         Generate<_T, _Dim>(test_points, _N, boundary);
185         NNSInstance<_T, _Dim, _N> perfect(boundary, test_points);
186         perfect.run(Algorithm::Naive, range_list[i]);
187
188         for ( int k = 0; k < Algorithm::Max; ++ k )
189             test_map.emplace(k, this->lauch_test(test_points, range_list[i], k, perfect));
190         vector_ref.emplace_back( std::move(test_map) );
191     }
192 }
193
194 /* For range 8. */
195 auto & vector8_ref = results[ 3 ];
196 for ( int i = 0; i < Algorithm::Max; ++ i ) {
197     std::cout << Algorithm::Text[i] << "_Range8Time = [ ";
198     for ( const auto & it : vector8_ref )
199         std::cout << std::get<0>( it.at(i) ) << ", ";
200     std::cout << "]" << std::endl;
201     std::cout << Algorithm::Text[i] << "_Range8Acc = [ ";
202     for ( const auto & it : vector8_ref )
203         std::cout << std::get<1>( it.at(i) ) << ", ";
204     std::cout << "]" << std::endl;
205 }
206
207 for ( int i = 0; i < Algorithm::Max; ++ i ) {
208     std::cout << Algorithm::Text[i] << "_AvgTime = [ ";
209     for ( size_t j = 0; j < range_list.size(); ++j ) {
210         double avg_time = 0.;
211         int counter = 0;
212         for ( const auto & it : results[j] ) {
213             avg_time += std::get<0>( it.at(i) );
214             counter++;
215         }
216         std::cout << avg_time / static_cast<double>(counter) << ", ";
217     }
218     std::cout << "]" << std::endl;
219     std::cout << Algorithm::Text[i] << "_AvgAcc = [ ";
220     for ( size_t j = 0; j < range_list.size(); ++j ) {
221         double avg_acc = 0.;
222         int counter = 0;
223         for ( const auto & it : results[j] ) {
224             avg_acc += std::get<1>( it.at(i) );
225             counter++;
226         }
227         std::cout << avg_acc / static_cast<double>(counter) << ", ";
228     }
229     std::cout << "]" << std::endl;
230 }
231 }
232
233 };
234
235 }

```

```
238     int main()
239     {
240         std::cout.precision(4);
241         std::cout << std::fixed;
242
243         using namespace SPHAlgorithms::NNS;
244
245         TestNNS<double, 3, 100> test(
246             Boundary<double, 3>{
247                 std::make_pair( Point<double, 3>{ 0., 0., 0.},
248                               Point<double, 3>{50., 50., 50.}
249             )
250         };
251         test.test();
252         return 0;
253     }
254
255
```

ВІДОМІСТЬ АТЕСТАЦІЙНОЇ РОБОТИ

Позначення	Найменування	Дод. відомості
	Текстові документи	
1	Пояснювальна записка	81 с.
2	Презентаційний матеріал	25 с.
	Інші документи	
3	Роздруківки програм	20 с.
4	Рецензія	2 с.
5	Відгук керівника	1 с.

					Оптимізація пошуку сусідів у методі гідродинаміки згладжених частинок					
Змін	Арк.	Номер докум.	Підп.	Дата	(Тема роботи) Відомість атестаційної роботи				Аркуш	Аркушів
Розроб.	Стрекозов А.Д.									
Перевір.	Артюх А.В.									
Н. контр.	Сидоров М.В.					ХНУРЕ				
Затв.	Гевяшев А.Д.				Кафедра ПМ					