CS 4414: Recitation 7

Sagar Jha

Today: Templates and Sokoban Part II

1. Templates (Presentation of Chapter 6 of *A Tour of C++*)
   - Parameterized Types : Vector<T> implementation, value template arguments, template argument deduction
   - Parameterized Operations: function templates, function objects, lambda expressions
   - Template mechanisms: variable templates, aliases, compile-time if
2. Sokoban Part II
   - Review the C++-17 implementation and code structure
   - Study the effects of filtering dead states on performance
   - Debug errors I fixed during the implementation using gdb

Templates: Introduction

- A class or function that we parameterize with a set of types or values
- Used to express general ideas independent of the types involved
- Templates plus the template arguments specify the complete class/function (template instantiation or specialization)
- Compiler generates proper classes or functions from their template specifications – thus, templates don’t have any runtime overheads
- e.g. From a definition of vector<T>, the user can create objects of both vector<int> and vector<double>. The compiler will generate code for vector<int> and vector<double> separately replacing T by int and double.

Parameterized Types: class Vector<T>

```
template<typename T>
class Vector {
private:
     T* elem;  // elem points to an array of sz elements of type
T
     int sz;
public:
     explicit Vector(int s);          // constructor: establish
invariant, acquire resources
     ~Vector() { delete[] elem; }     // destructor: release
resources

     // ... copy and move operations ...

     T& operator[](int i);                       // for non-const
Vectors
     const T& operator[](int i) const;           // for const
Vectors (§4.2.1)
     int size() const { return sz; }
};
```

Parameterized Types: class Vector<T>

- Defining objects of type Vector<T>

```
Vector<char> vc(200);          // vector of 200 characters
Vector<string> vs(17);         // vector of 17 strings
Vector<list<int>> vli(45);     // vector of 45 lists of integers
```

- Using a Vector<string> object

```
void write(const Vector<string>& vs)        // Vector of some strings
{
    for (int i = 0; i!=vs.size(); ++i)
        cout << vs[i] << '\n';
}
```

Parameterized Types: class Vector<T>

- Implementation of member functions

```
template<typename T>
Vector<T>::Vector(int s)
{
     if (s<0)
          throw Negative_size{};
     elem = new T[s];
     sz = s;
}

template<typename T>
const T& Vector<T>::operator[](int i) const
{
     if (i<0 || size()<=i)
          throw out_of_range{"Vector::operator[]"};
     return elem[i];
}
```

Parameterized Types: class Vector<T>

- Implementation of begin and end for iteration

```
template<typename T>
T* begin(Vector<T>& x)
{
    return x.size() ? &x[0] : nullptr;    // pointer to first element or nullptr
}

template<typename T>
T* end(Vector<T>& x)
{
    return x.size() ? &x[0]+x.size() : nullptr;    // pointer to one-past-last element
}
```

Value Template Arguments

- A template can take value arguments, in addition to type arguments
- From STL, we have std::array<T, N> where N is the size of the array
- e.g. Buffer<T, N>

```
template<typename T, int N>
struct Buffer {
    using value_type = T;
    constexprint size() { return N; }
    T[N];
    // ...
};
```

Template Argument Deduction

- **auto p = make_pair(1, 5.2);** or **pair p = {1, 5. 2};** – Compiler will deduce the type of p to be pair<int, double>

```
template<typename T>
class Vector {
public:
        Vector(int);
        Vector(initializer_list<T>);        // initializer-list constructor
        // ...
};

Vector v1 {1,2,3};      // deduce v1's element type from the initializer element type
Vector v2 = v1;         // deduce v2's element type from v1's element type

auto p = new Vector{1,2,3};     // p points to a Vector<int>

Vector<int> v3(1);      // here we need to be explicit about the element type (no element type is mentioned)
```

Function Templates – Generic sum function

- Functions can also be templated

```
template<typename Sequence, typename Value>
Value sum(const Sequence& s, Value v)
{
     for (auto x : s)
          v+=x;
     return v;
}
```

- The algorithm's library function std::accumulate() provides a general version of sum

Function Templates – Generic sum function

```
void user(Vector<int>& vi, list<double>& ld,
vector<complex<double>>& vc)
{
     int x = sum(vi,0);                       // the sum of a
vector of ints (add ints)
     double d = sum(vi,0.0);                  // the sum of a
vector of ints (add doubles)
     double dd = sum(ld,0.0);                 // the sum of a list
of doubles
     auto z = sum(vc,complex{0.0,0.0});       // the sum of a
vector of complex<double>s
}
```

Function objects (or functor)

- In C++, you can overload operator() as **Ret operator()(Args… args);**

```
template<typename T>
class Less_than {
     const T val;   // value to compare against
public:
     Less_than(const T& v) :val{v} { }
     bool operator()(const T& x) const { return x<val; } // call
operator
};
```

Function objects (or functor)

```
Less_than lti {42};              // lti(i) will compare i to
42 using < (i<42)
Less_than lts {"Backus"s};       // lts(s) will compare s to
"Backus" using < (s<"Backus")
Less_than<string> lts2 {"Naur"};   // "Naur" is a C-style
string, so we need <string> to get the right <
```

```
void fct(int n, const string& s)
{
      bool b1 = lti(n);    // true if n<42
      bool b2 = lts(s);    // true if s<"Backus"
      // ...
}
```

Using function objects as predicates

```
template<typename C, typename P>
     // requires Sequence<C> && Callable<P,Value_type<P>>
int count(const C& c, P pred)
{
     int cnt = 0;
     for (const auto& x : c)
           if (pred(x))
                 ++cnt;
     return cnt;
}
```

```
void f(const Vector<int>& vec, const list<string>& lst, int x,
const string& s)
{
     cout << "number of values less than " << x << ": " <<
count(vec,Less_than{x}) << '\n';
     cout << "number of values less than " << s << ": " <<
count(lst,Less_than{s}) << '\n';
}
```

Lambda Expressions

- Notation for implicitly generating function objects

```
void f(const Vector<int>& vec, const list<string>& lst, int x,
const string& s)
{
     cout << "number of values less than " << x
          << ": " << count(vec,[&](int a){ return a<x; })
          << '\n';
     cout << "number of values less than " << s
          << ": " << count(lst,[&](const string& a){ return a<s;
})
          << '\n';
}
```

Lambda Expressions : for_all function

```
template<typename C, typename Oper>
void for_all(C& c, Oper op)        // assume that C is a
container of pointers
     // requires Sequence<C> && Callable<Oper,Value_type<C>>
(see §7.2.1)
{
     for (auto& x : c)
          op(x);       // pass op() a reference to each element
pointed to
}
```

```
template<class S>
void rotate_and_draw(vector<S>& v, int r)
{
     for_all(v,[](auto& s){ s->rotate(r); s->draw(); });
}
```

Template Mechanisms

- Variables can also be templated

```
template <class T>
     constexpr T viscosity = 0.4;

template <class T>
     constexpr space_vector<T> external_acceleration = { T{},
T{-9.8}, T{} };

auto vis2 = 2*viscosity<double>;
auto acc = external_acceleration<float>;
```

Template Mechanisms

- Aliases allow us to use types related to template arguments

```
template<typename T>
class Vector {
public:
      using value_type = T;
      // ...
};
```

```
template<typename C>
using Value_type = typename C::value_type;          // the type of C's elements

template<typename Container>
void algo(Container& c)
{
      Vector<Value_type<Container>> vec;            // keep results here
      // ...
}
```

Template Mechanisms

- Aliasing can be used to bind some or all template arguments

```
template<typename Key, typename Value>
class Map {
     // ...
};

template<typename Value>
using String_map = Map<string,Value>;

String_map<int> m;     // m is a Map<string,int>
```

Template Mechanisms

- Not all code can be general
- *Compile-Time if* combined with type traits can help

```
template<typename T>
void update(T& target)
{
    // ...
    if constexpr(is_pod<T>::value)
        simple_and_fast(target); // for "plain old data"
    else
        slow_and_safe(target);
    // ...
}
```

Why can templates only be implemented in the header file?

- NOT TRUE, but often necessary
- Write the implementations in an associated .hpp file (my_template_library_impl.hpp), then include this file at the end of the header file (my_template_library.hpp)
- If you know the template instantiations beforehand, alternatively declare them in the .cpp file (my_template_library.cpp)
- Read https://stackoverflow.com/questions/495021/why-can-templates-only-be-implemented-in-the-header-file

Part II : Sokoban Part II

Recap:

- Learned to play the game of Sokoban
- Reviewed versions 1, 2, 3, 4 of Sokoban solver
- Learned the trade-offs between DFS & BFS for game tree exploration
- Main optimization idea: Exploration on box moves, not player moves
- Surveyed the 15 test cases to gain an understanding of solving them computationally
- Fixed bug in high memory usage: allocating objects using new is an invitation to leaking memory
- v3_opt performs best time-wise and uses the least amount of memory

Sokoban Part II

Issues:

- The best solution still takes about 90 seconds for solving level 10
- Tracing moves made while exploring state is inefficient
- Time to solve and memory usage is directly proportional to the number of states explored before finding a solution. Need to reduce this number to optimize both

Today:

- A modern implementation using C++-17 (including compiling and linking)
- Improved tracing and class design
- Filtering dead states to optimize search time and memory usage
- Reachability analysis for finding all box moves
- Using gdb to catch and fix bugs

Code organization and building

- Files
  - ./main.cpp: Includes code to input a sokoban puzzle and output the solution
  - ./sokoban/sokoban.hpp: Declares classes sokoban_solver, sokoban_state and sokoban_board
  - ./sokoban/sokoban.cpp: Defines member functions of classes in sokoban.hpp
- Building/compiling
  - Using CMake – CMakeLists.txt in both ./ and ./sokoban
  - Different build types Release and Debug with different g++ options
  - sokoban.hpp and sokoban.cpp are compiled to the library libsokoban_solver.so
  - main.cpp is linked to libsokoban_solver.so to generate the binary sokoban

Class Design

- class *sokoban_solver* implements BFS to find a solution
- class *sokoban_state* stores an intermediate state or configuration
- *sokoban_state* finds all moves that can be made from an instance
- Observation: Every sokoban state that is explored contains the same board texture – Empty spaces and Walls
- Idea: Refactor the board texture and use the same instance across all states explored from a given puzzle
- Approach: Carefully navigate the ownership of the shared board texture object

Internals

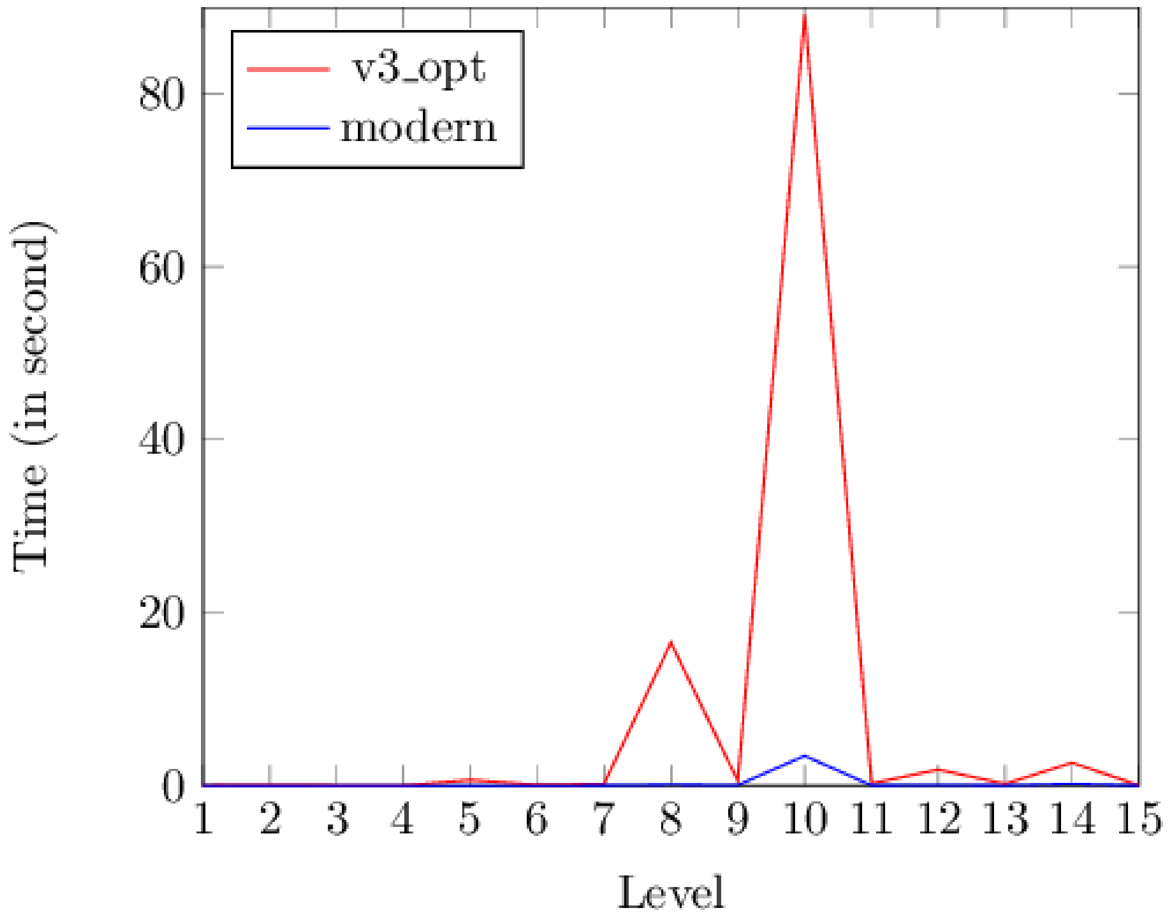
- Board texture
  - Flattened 1D-array for efficiency
- Tracing
  - Each state stores a vector of moves made to reach that state
- Reachability analysis (to compute all one box moves)
  - Mark the player position as reachable, box positions as unreachable
  - Expand the reachable positions iteratively by trying to move down, up, right or left from reachable positions
  - A box can be moved down, if the square above it is reachable and the square below it is an empty square (or floor). Similarly, for up, right and left.
  - Iterate through all boxes to find all the moves

Main optimization for both runtime and memory usage : Filtering dead states

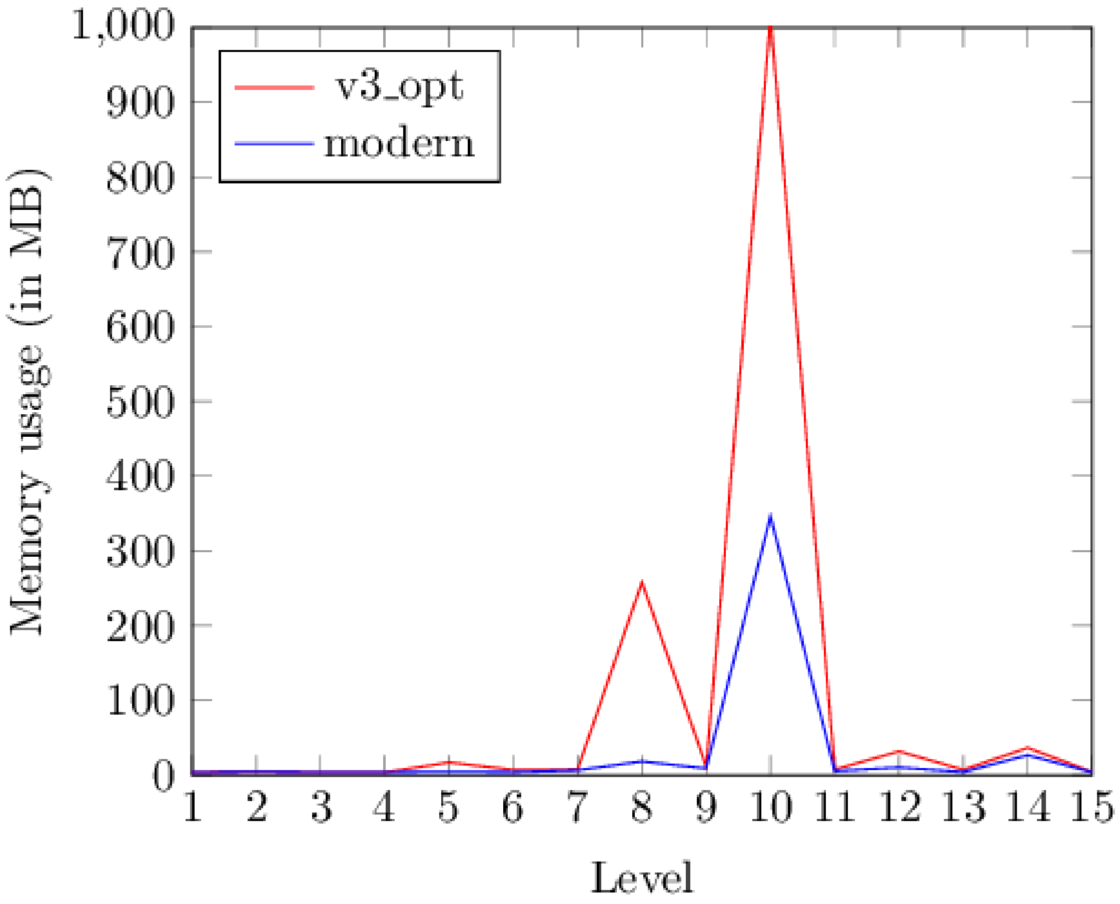
- Sometimes, it is possible to examine a state and tell that a solution can never be reached from there
- Immovable box: If a box cannot be moved either down/up/right/left and it’s not already on a target square, no solution is possible
- Boxes along an edge: If more boxes than targets exist along an edge, no solution is possible
- Stacked boxes: Boxes next to each other can block each other resulting in dead states

Aside: Bash script for evaluation and latex files for plotting

Immovable box filter gives up to 150X time improvement and uses up to 15X less memory!

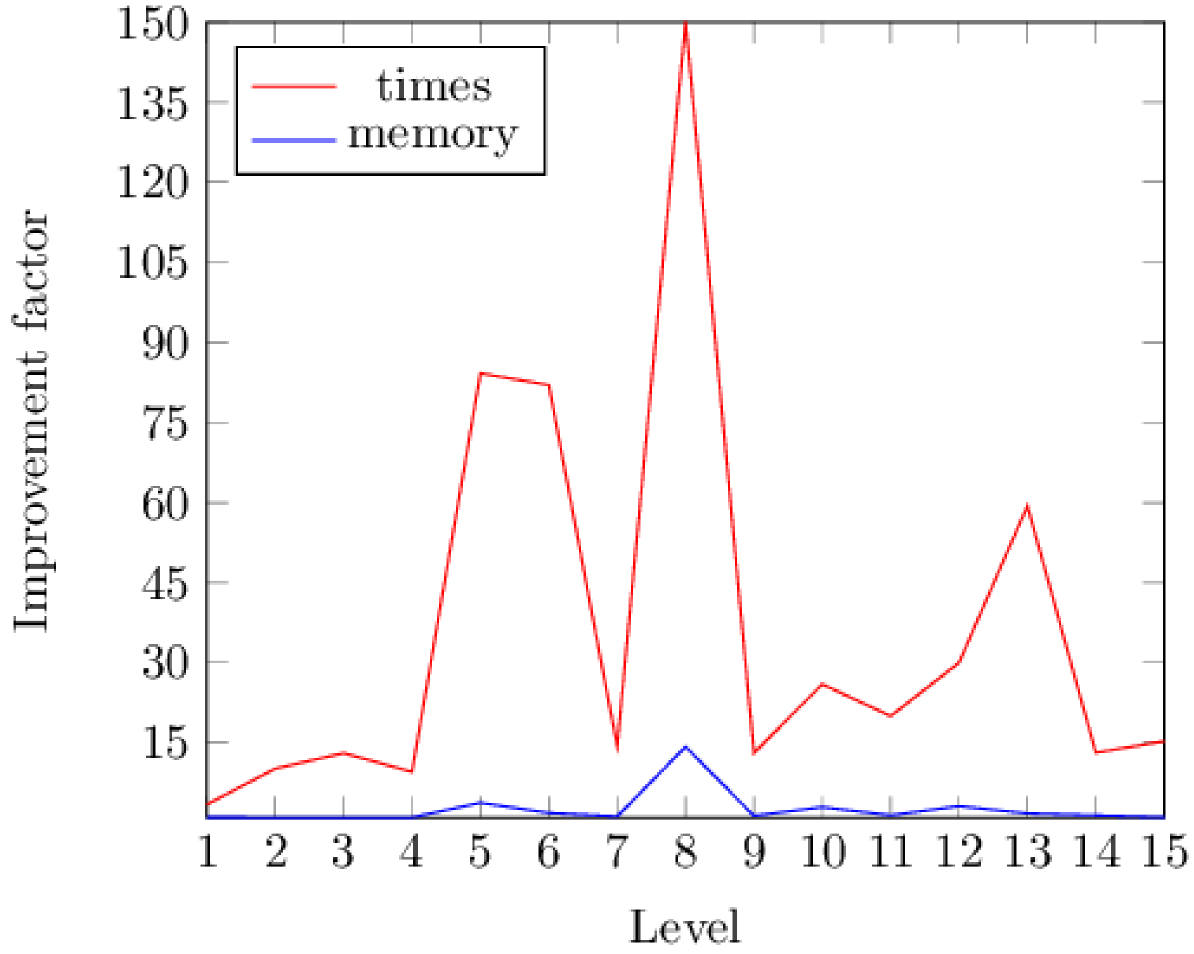


Time to solve vs level



Memory usage vs level

Immovable box filter gives up to 150X time improvement and uses up to 15X less memory!



Improvement factor for modern code over v3_opt

Debugging sokoban with gdb

- Faulty shadowing: I often name local variables same as class variables
  - Bug cause: An object can be passed to its own constructor!
  - See https://stackoverflow.com/questions/32608458/is-passing-a-c-object-into-its-own-constructor-legal
- Bug in reachability: Forgot to check the board texture
- High memory usage: Permutations of box positions are equivalent
- Tracing bug: After permutations are made equivalent, tracing becomes faulty

Exercise for the more interested

- Extend my solution using templates to write a general one-player game solver