Implementing Fast Heuristic Search Code

Etan Burns, Matthew Hatem, Michael J. Jeighton and Wheeler Ruml

Presentation: Christopher Scherb
Overview

- Optimizations for search algorithms are tested
- State-of-the-art implementation of a heuristic search
- Test implementation of an A* and an IDA* algorithm
- Test scenario: 15-puzzle
- Try different optimizations:
  - Programming language based optimizations
  - Algorithm based optimizations
Measure of the performance

• A complexity analysis predicts the performance of an algorithm
• Number of expanded nodes does not account the computation overhead
• No good theory for the implementation
• Focus on solving time
Why use optimized code?

• Difference between fast and slow implementation can be significant
• Non-optimized algorithms can lead to wrong conclusions
• Get familiar with state-of-the-art tricks
• But: Better algorithm will outperform code optimizations
Test scenarios

• 15-puzzle: popular benchmark
  – 1,046,139,494,400 reachable states
  – Unique path costs
• 100 different start configurations
• Algorithms: A* and IDA*
• Heuristic: Manhattan distance
IDA*

• Iterative depth-first search
• Tree search
• Memory efficient
• The iterative depth overhead can often be ignored
• Limited on f values, with $f = g + h$
function recursive-search(n, f-limit):
    if f(n) > f-limit:
        return <f(n), none>
    if is-goal(n.state):
        return <none, solution(n)>
    next-limit = inf
    for each <a, s'> in succ(n.state):
        if h(s') < inf:
            n' = make-node(n,a,s')
            <rec-limit, solution> := recursive-search(n', f-limit)
            if solution != none:
                return <none, solution>
            next-limit := min(next-limit, rec-limit)
    return <next-limit, none>
Base Implementation IDA*

• Implemented in C++
• Search domain: Black box
  • Initial
  • Isgoal
  • Expand
  • Release
• Can run on each search domain
  – Search domain implements an interface
  – Virtual functions are necessary
Base Implementation IDA*

**SearchAlgorithm**
- \&dom: SearchDom
- search(initState): path

**SearchDom**
- init(): SearchState
- h: int
- isgoal: bool
- expand(SearchState): Edge[]
- release()

**SearchState**
- blank: int
- h: int
- tiles: int[]
- isgoal(): bool
- release()

**15puzzle**
- init(): SearchState
- h: int
- isgoal: bool
- expand(SearchState): Edge[]
- release()
Base Implementation IDA*

– Operator order:
  • Up, left, right, down
  • Possibly very important

```cpp
std::vector expand(SearchState s) {
    std::vector<Kid> kids;
    if(s.blank >= width) // use blank filed up
        kids.push(kid(s, s.blank - Width));
    ... // same for all other directions
    return kids;
}

Kid kid(SearchState s, int newblank) {
    SearchState kid = new SearchState();
    kid.tiles = s.tiles.copy();
    kid.tiles[s.blank] = kid.tiles[newblank];
    kid.blank = newblank;
    kid.h = Manhattan-Dist(kid.tiles, kid.blank);
    return kid;
}
```
Incremental Manhattan Distance

• Compute the Manhattan Distance incrementally.
  – Compute the Manhattan Distance for the first state.
  – Store cost change for each possible single move → lookup table
  – Heuristic: just look in the table and add the value
Operator Pre-computation

• Pre-compute the legal successor positions for each 16 possible blank locations before searching
• Store legal successor positions
  – Operator Table
• Choose only the legal successors
In-place modification

• Copy of the state vector is expensive
  – Copy an array of size 16

• Better: in-place modification
  – Applying operation and undoing is cheaper
  – More memory efficient
  – Quite likely that one state remains in the cache

• Undo requires information of the parent node:
  – Heuristic value
  – Blank position
C++ Templates

• Calling a virtual method is expensive
• Alternative: C++ Templates
  – Meta programming
  – Similar to a find and replace
  – Methods are known when compiling
• Additional advantage:
  – No opaque pointers necessary
  – No dynamic memory allocation necessary
C++ Templates

```cpp
struct Tiles {
    struct State {
        ...
    } State initial() {
        ...
    }
    int h(const State s&)
    {
    }
}

template<class D> struct SearchAlg {
    SearchAlg(D &d) : dom(d) { }
    virtual std::vector<typename D::State> search(typename D::State&) = 0;
    D &dom;
}
int main(int argc, char **argv) {
    ... SearchAlg<Tiles> search;
    ...
```
## Time

<table>
<thead>
<tr>
<th></th>
<th>secs</th>
<th>imp. to base</th>
<th>Nodes/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base implementation</td>
<td>9,298</td>
<td>-</td>
<td>1,982,479</td>
</tr>
<tr>
<td>Incremental heuristic</td>
<td>5,476</td>
<td>1.7</td>
<td>3,365,735</td>
</tr>
<tr>
<td>Oper. Pre-computation</td>
<td>5,394</td>
<td>1.7</td>
<td>3,417,218</td>
</tr>
<tr>
<td>In-place modification</td>
<td>2,179</td>
<td>4.3</td>
<td>8,457,031</td>
</tr>
<tr>
<td>C++ Templates</td>
<td>634</td>
<td>14.7</td>
<td>29,074,838</td>
</tr>
<tr>
<td>Direct C implementation</td>
<td>666</td>
<td>14.0</td>
<td>27,637,121</td>
</tr>
</tbody>
</table>

• Note: no optimization was removed.
Summary IDA*

• Programming language based optimizations:
  – C++ Templates

• Algorithm based optimizations:
  – Incremental Manhattan Distance
  – Operator pre-computation
  – In-place modifications
  – Operator order
A*

- Breadth-first search
- Large memory requirements
- $f = g + h$
- Open list: stores nodes to expand
- Closed list: stores already expanded nodes
A*

open := new min. heap ordered by f
open.insert(make-root-node(init()))
distances := new hash-table
while not open.empty():
    n = open.pop-min()
    if n.state not in distances or g(n) < distances[n.state]:
        distances[n.state] := g(n)
    if is.goal(n.state):
        return extract-solution(n)
    for each <a,s'> in succ(n.state):
        if h(s') < inf:
            n' := make-node(n,a,s')
            open.insert(n')

return unsolvable
Base Implementation A∗

• Open list: binary min-heap
  – Allow duplicated nodes in the open list
• Closed list: Hash table
• Memory-Limit: 46GB
  – First run exhausted memory in 3 of the 100 tests
Previous Optimizations

• In-Place modification not possible
• Incremental Manhattan Distance
• Oper. Pre-computation
• Only small advantage
  – Spends much time on open list and closed list
  – Hash value and Operator calculation have only a small influence
Detecting duplicates on the open list

• One of the most common optimization
• Only a single node for each state
• Update node on the open list if there is cheaper path
• → each state becomes a single canonical node
• Detecting duplicates takes a long time
• More beneficial for other search problems
C++ Templates

• Were very beneficial for IDA*
  – Do not need any dynamic memory allocation
• Reduce amount of memory allocations
• Only one of 100 problems cannot be solved
Pool Allocation

• A* does a lot of malloc operations
• Memory allocation:
  – Better to allocate a small number of large chunks
  – Allocates 1024 nodes at the same time
• Reduces the number of expensive heap allocations
• Can solve all 100 problems
Packed State Representations

• Optimize the memory usage
• Numbers 1-15 can be represented by 4 bits
• \(4\ \text{bits} \times 16\ \text{tiles} = 8\ \text{bytes}\) for a state
  – Instead of 1 byte for each tile = 16 bytes for a state
  – Half the memory requirements
• Closed list operation requires hashing a 16-entry array
Intrusive Data Structures

• Hash table resolves collisions with chaining
• Two pointers, one to the entry, one to the next element
• Intrude the next element pointer to the hash table entry
• Reduce memory requirement by 16 bytes per node (64bit OS)
Array based Priority Queues

• Take advantage of unit cost edges
• 1-level bucket priority queue
  – Replaces binary heap
  – Simply array indexed by f values
  – Constant time for insert and remove
• Nested bucket priority queue
  – Uses a 2-level bucket priority queue
  – Inner bucket queue sorts by g-values
## Time (97 problems)

<table>
<thead>
<tr>
<th>Implementation</th>
<th>secs</th>
<th>imp.</th>
<th>GB</th>
<th>Imp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base implementation</td>
<td>1,695</td>
<td>-</td>
<td>28</td>
<td>-</td>
</tr>
<tr>
<td>Incr. MD and oper. table</td>
<td>1,513</td>
<td>1.1</td>
<td>28</td>
<td>1.0</td>
</tr>
<tr>
<td>Avoiding dups on open</td>
<td>1,968</td>
<td>0.9</td>
<td>28</td>
<td>1.0</td>
</tr>
<tr>
<td>C++ templates</td>
<td>1,273</td>
<td>1.3</td>
<td>23</td>
<td>1.2</td>
</tr>
<tr>
<td>Pool allocation</td>
<td>1,184</td>
<td>1.4</td>
<td>20</td>
<td>1.4</td>
</tr>
<tr>
<td>Packed states</td>
<td>1,051</td>
<td>1.6</td>
<td>18</td>
<td>1.6</td>
</tr>
<tr>
<td>Intrusive data structures</td>
<td>709</td>
<td>2.4</td>
<td>15</td>
<td>1.9</td>
</tr>
<tr>
<td>1-level bucket open list</td>
<td>450</td>
<td>3.8</td>
<td>21</td>
<td>1.3</td>
</tr>
<tr>
<td>2-level bucket open list</td>
<td>290</td>
<td>5.8</td>
<td>11</td>
<td>2.5</td>
</tr>
</tbody>
</table>
## Time (100 problems)

<table>
<thead>
<tr>
<th></th>
<th>secs</th>
<th>GB</th>
<th>nodes/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pool allocation</td>
<td>2,375</td>
<td>45</td>
<td>656,954</td>
</tr>
<tr>
<td>Packed states</td>
<td>1,896</td>
<td>38</td>
<td>822,907</td>
</tr>
<tr>
<td>Intrusive data structures</td>
<td>1,269</td>
<td>29</td>
<td>1,229,574</td>
</tr>
<tr>
<td>1-level bucket open list</td>
<td>727</td>
<td>36</td>
<td>3,293,048</td>
</tr>
<tr>
<td>Nested bucket open list</td>
<td>516</td>
<td>27</td>
<td>3,016,135</td>
</tr>
</tbody>
</table>
Summary A*

• Programming language based optimizations:
  – Pool allocation and packed states can be used in each implementation
  – Use C++ Templates if available

• Algorithm based optimizations:
  – Incremental Heuristic
  – Intrusive data structures mildly beneficial
  – Bucket Queues may be an advantage
Notes

• Benchmarking State-of-the-art implementations is problematic
• Optimizations are important to determine empirically which implementation is faster
• 15-puzzle problematic benchmark scenario
Questions?

Thank you for your attention
### Case Study

- **Objective Caml**
- **High level language**
- **Garbage collection**

<table>
<thead>
<tr>
<th></th>
<th>secs / C++</th>
<th>secs / Obj. Caml</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base implementation</td>
<td>9,298</td>
<td>-</td>
</tr>
<tr>
<td>Incremental heuristic</td>
<td>5,476</td>
<td>24,186</td>
</tr>
<tr>
<td>Oper. Pre-computation</td>
<td>5,394</td>
<td>-</td>
</tr>
<tr>
<td>In-place modification</td>
<td>2,179</td>
<td>3,261</td>
</tr>
<tr>
<td>C++ Templates</td>
<td>634</td>
<td>-</td>
</tr>
<tr>
<td>Direct implementation</td>
<td>666</td>
<td>852</td>
</tr>
</tbody>
</table>