Implementing Fast Heuristic Search Code

E. Burns, M. Hatem, M. J. Leighton, W. Ruml
(presentation by G. Röger)

Universität Basel

October 17, 2013

Motivation and Background

Motivation

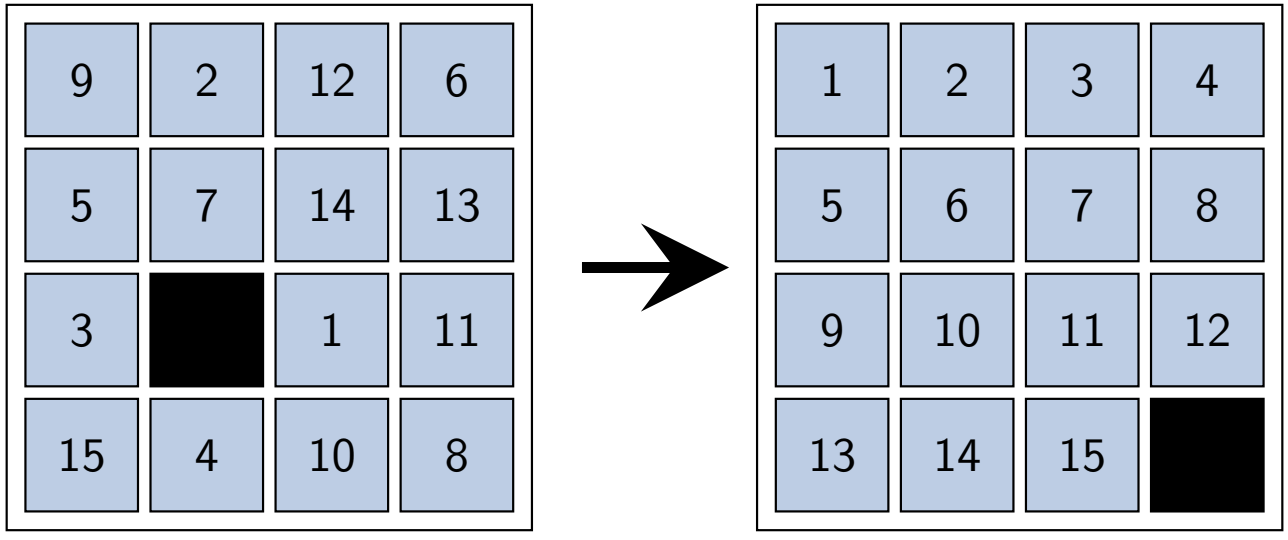
- Theoretical comparison of different search methods often hard/impossible
- Very often: Empirical comparison by means of experiments
- Performance depends on implementation
- Usually no implementation details in publications

Aim of the paper

Example:
A* and IDA* in C++
with Manhattan distance heuristic on 15-puzzle

- Demonstrate influence of implementation
- Reveal "tricks" of state-of-the-art implementations and measure their influence
- Focus on solving time

Reminder: Manhattan Distance Heuristic



Manhattan distance $MD_t(s)$:
Distance from position of tile t in s to goal position of t
Example: $MD_1(s) = 4$

Manhattan distance heuristic $h^{\mathrm{MD}}(s) = \sum_{\text{Tiles } t} MD_t(s)$

IDA*

Base Implementation

Domain-independent IDA* implementation

- Virtual methods `Initial`, `h`, `IsGoal`, `Expand`, `Release`
- Opaque states

```
IDAStar(initial configuration):
    state = Initial(initial configuration)
    f-limit = 0
    while f-limit != infinity:
        f-limit = LimitedDFS(state, 0, f-limit)
    return unsolvable
```

Base Implementation

```
LimitedDFS(state, g, f-limit):
    if g + h(state) > f-limit then
        return g + state.h
    if IsGoal(state) then
        extract solution and stop search
    next-limit = infinity
    for each child in Expand(state) do
        rec-limit = LimitedDFS(child, g + 1, f-limit)
        Release(child)
        next-limit = min(next-limit, rec-limit)
    return next-limit
```

Base Implementation

Sliding tile puzzle

- State consists of
  - field `blank` for blank position
  - field `h` for heuristic estimate
  - array `tiles` with 16 integers maps positions to tiles
- `h`, `IsGoal` and `Release` simple one-liners
- `Initial` creates initial state as specified in input
- Dynamics of domain in method `Expand`

Base Implementation: Expand

```
Expand (State s):
  vector childs // creates an empty vector
  if s.blank >= Width then
    childs.push(Child(s, s.blank - Width))
  if s.blank % Width > 0 then
    childs.push(Child(s, s.blank - 1))
  if s.blank % Width < Width - 1 then
    childs.push(Child(s, s.blank + 1))
  if s.blank < NTiles - Width then
    childs.push(Child(s, s.blank + Width))
  return childs
```

Base Implementation: Child

```
Child (State s, int newblank):
  State child = new State
  Copy s.tiles to child.tiles
  child.tiles[s.blank] = s.tiles[newblank]
  child.blank = newblank
  child.h = Manhattan_dist(child.tiles, child.blank)
  return child
```

Base Implementation: Performance

Solves all 100 benchmark tasks in 9 298 seconds.

1. Improvement: Incremental Manhattan Distance

- State s with tile t^* at position *from_pos*
- Successor state s' with tile t^* at position *to_pos*
- Only MD of tile t^* changes

$$h^{\mathrm{MD}}(s') = \sum_{\text{Tiles } t} MD_t(s')$$
$$= h^{\mathrm{MD}}(s) - MD_{t^*}(s) + MD_{t^*}(s')$$

- Idea: Precompute MD difference of tile t^* when moving it from *from_pos* to *to_pos*
- Lookup table `MDInc[tile][from_pos][to_pos]`

1. Improvement: Incremental Manhattan Distance

```
Child (State s, int newblank):
  State child = new State
  Copy s.tiles to child.tiles
  child.tiles[s.blank] = s.tiles[newblank]
  child.blank = newblank
  child.h = Manhattan_dist(child.tiles, child.blank)
  return child
```

→ Solves all instances in 5 476 seconds (previously 9 298 seconds)

1. Improvement: Incremental Manhattan Distance

```
Child (State s, int newblank):
  State child = new State
  Copy s.tiles to child.tiles
  child.tiles[s.blank] = s.tiles[newblank]
  child.blank = newblank
  child.h = s.h + MDInc[s.tiles[newblank]][newblank][s.blank]
  return child
```

→ Solves all instances in 5 476 seconds (previously 9 298 seconds)

2. Improvement: Operator Pre-computation

```
Expand (State s):
  vector childs // creates an empty vector
  if s.blank >= Width then
    childs.push(Child(s, s.blank - Width))
  if s.blank % Width > 0 then
    childs.push(Child(s, s.blank - 1))
  if s.blank % Width < Width - 1 then
    childs.push(Child(s, s.blank + 1))
  if s.blank < NTiles - Width then
    childs.push(Child(s, s.blank + Width))
  return childs
```

Idea: Avoid evaluations in if statements by precomputing possible movements

2. Improvement: Operator Pre-computation

Pre-compute `applicable_ops[blank_pos]`
→ array of possible next blank positions

```
Expand (State s):
  vector childs
  for newblank in applicable_ops[s.blank] do
    childs.push(Child(s, newblank))
  return childs
```

→ Solves all instances in 5 394 seconds (previously 5 476 seconds)

3. Improvement: In-place Modification

```
Child (State s, int newblank):
  State child = new State
  Copy s.tiles to child.tiles
  ...
  return child
```

Problem: Copying takes lots of time (and also memory)
Idea: Modify state and revert it when backtracking

3. Improvement: In-place Modification (Conceputally)

```
LimitedDFS(state, g, f-limit):
    ...
    for each child in Expand(state) do
        rec-limit = LimitedDFS(child, g + 1, f-limit)
        Release(child)
    ...
```

3. Improvement: In-place Modification (Conceputally)

```
LimitedDFS(state, g, f-limit):
    ...
    for each child in Expand(state) do
        rec-limit = LimitedDFS(child, g + 1, f-limit)
        Release(child)
    ...
```

```
LimitedDFS(state, g, f-limit):
    ...
    for i = 0 to NumOfApplicableOps(state) do
        undoinfo = ApplyNthOp(state, i)
        rec-limit = LimitedDFS(state, g + 1, f-limit)
        Undo(state, undoinfo)
    ...
```

3. Improvement: In-place Modification (15-puzzle)

```
ApplyNthOp(state, n):
    u = new UndoInfo()
    u.h = state.h
    u.blank = s.blank
    newblank = applicable_ops[s.blank][n]
    tile = state.tiles[newblank]
    state.h += MDInc[tile][newblank][state.blank]
    state.tiles[state.blank] = tile
    state.blank = newblank
    return u
```

3. Improvement: In-place Modification (15-puzzle)

```
ApplyNthOp(state, n):
    u = new UndoInfo()
    u.h = state.h
    u.blank = s.blank
    newblank = applicable_ops[s.blank][n]
    tile = state.tiles[newblank]
    state.h += MDInc[tile][newblank][state.blank]
    state.tiles[state.blank] = tile
    state.blank = newblank
    return u

Undo(state, undoinfo):
    state.tiles[s.blank] = state.tiles[undoinfo.blank]
    state.h = undoinfo.h
    state.blank = undoinfo.blank
    delete undoinfo
```

3. Improvement: In-place Modification

$\rightarrow$ Solves all instances in 2 179 seconds (previously 5 394 seconds)

4. Improvement: C++ Templates

- Main problem: Virtual methods cannot be inlined
- Solution: Use templates
- Additional advantage: no opaque pointers
- Resulting machine code same as from pure sliding-tiles solver implementation

$\rightarrow$ Solves all instances in 634 seconds (previously 2 179 seconds)

IDA* Summary

Base implementation	9,298	–	1,982,479
Incremental heuristic	5,476	1.7	3,365,735
Operator pre-computation	5,394	1.7	3,417,218
In-place modification	2,179	2.3	8,457,031
C++ template	634	14.7	29,074,838
Korf's solver	666	14.0	27,637,121

A*

Base Implementation

Standard A* implementation

- Open list: binary min-heap ordered on f
  (tie-breaking prefers high g)
- Allows duplicate states in open list
- Closed list: hash table using chaining to resolve collissions
- Positions and tiles in `State` no longer integers but bytes

$\rightarrow$ can solve only 97 of the 100 instances
$\rightarrow$ solving these 97 requires 1 695 seconds

1. Improvement: Detecting Duplicates on Open

When pushing a new node to open:

- Is there already a node with the same state in Open?
- If not, add the new node
- If yes and the new node has a lower g-value
  $\rightarrow$ update the node in Open
  - g-value
  - parent pointer
  - position in Open (according to new f-value)

Solving time for the 97 instances increases from 1 695 seconds to 1 968 seconds

2. Improvement: C++ Templates

- Changes analogous to IDA* case
- With IDA* no need for memory allocation during search
- A* must still allocate search nodes

Solving time for the 97 instances drops from 1 695 seconds to 1 273 seconds and two more instances solved.

3. Improvement: Pool Allocation

- Memory consumption of A* grows during execution.
- Heap memory allocation takes time
- Idea: Do not allocate memory for one node after the other but for 1024 blocks at a time
- Positive side effect: increased cache locality

→ Solving time for the 97 instances drops to 1 184 seconds
→ Solves all instances within 2,375 seconds

4. Improvement: Packed State Representation

- Most time spent on open and closed list operations
- Closed list operation: hashing 16-entry array and possibly performing equality tests on it
- Improvement: Pack tiles into 8 bytes (1 word on many machines)

Benefits

- Less memory consumption
- Hash function is simple return
- Equality test is comparison of two numbers

$\rightarrow$ Solves all instances within 1,896 seconds (before: 2,375 seconds)

5. Improvement: Intrusive Data Structures

- Hash table resolves collisions by chaining
- Need in addition to the entry two pointers to previsous and next element
- Hash table contains a record for each element that adds these pointers.
- Idea: Avoid creating the records by allowing the hash table to add the pointers directly to the nodes (super-hackish)
- Also reduces memory requirement

→ Solves all instances within 1,269 seconds (before: 1,896 seconds)

6. Improvement: Array-based Priority Queues

- Min-heap open list: $O(\log_2 n)$ complexity for insert, remove and update
- Idea: Use array-based implementation instead
- Array holds at position i a list of all nodes with f-value i.
- Constant-time (amortized) inserting, removal and updating

$\rightarrow$ Solves all instances within 727 seconds (before: 1,269 seconds)

6. Improvement: Array-based Priority Queues (nested)

- Disadvantage of array-based priority queue: no g-value tie-breaking
- Solution: Nested open-list
- Bucket for f-value contains array-based priority queue sorted by g-value

$\rightarrow$ Solves all instances within 516 seconds (before: 727 seconds)

A*: Summary

97 initially solved instances

	secs	imp	GB	imp
Base implementation	1,695	–	28	–
Incr. MD and op. table	1,513	1.1	28	1.0
Avoid duplicates in open	1,968	0.9	28	1.0
C++ templates	1,273	1.3	23	1.2
Pool allocation	1,184	1.4	20	1.4
Packed states	1,051	1.6	18	1.6
Intrusive data structures	709	2.4	15	1.9
1-level bucket open list	450	3.8	21	1.3
Nested bucket open list	290	5.8	11	2.5

A*: Summary (all instances)

A*

	secs	GB	nodes/sec
Pool allocation	2,275	45	656,954
Packed states	1,896	38	822,907
Intrusive data structures	1,269	29	1,229,574
1-level bucket open list	727	36	3,293,048
nested bucket open list	516	27	3,016,135

IDA*

	secs	imp	nodes/sec
Base implementation	9,298	–	1,982,479
Incremental heuristic	5,476	1.7	3,365,735
Operator pre-computation	5,394	1.7	3,417,218
In-place modification	2,179	2.3	8,457,031
C++ template	634	14.7	29,074,838
Korf's solver	666	14.0	27,637,121

Conclusion

Conclusion

- Empirical comparison of algorithms has only limited informative value
- Importance of efficient implementation
- How helpful is this study?
  - only 15-puzzle (unit-cost invertible actions, cheap node expansion, few duplicates, ...)
  - cheap, rather uninformative heuristic
  - Which/how can improvements be generalized to other domains? Do findings still hold there?

Questions?

Questions?