Minimising overall jumps in CPS-style compilers

TIANYU LI

(supervisor: Dan Ghica)

MINIMISING OVERALL JUMPS IN CPS-STYLE COMPILERS

Abstract

This report will describe algorithms involved in minimising the amount of jumping for complier. A related mathematic model describing the architecture of program will be presented and dynamic programming will be used to analysis the problem. To simplify and proceed the project, this report will demonstrate the definition of implicit links using a generalized cost function to evaluate jumpings, which transfer the target of the problem into maximising the number of implicit links.

Then the specific problem could be transformed into traveling salesman problem (TSP), which was proof as NP-complete problem. Implementation and benchmarking with different algorithms will be presented in the report.

The programs written for this project can be found at:
https://codex.cs.bham.ac.uk/svn/projects/2015/txl486/

CONTENTS

1 Introduction

This report describes the work involed in minising overall jumps in CPS-style compiler. The goal of this project is: to present a mathematic model to evaluate and analysis the cost of jumps in assembly; to demonstrate potential algorithms minising overall jumps by re-ordering the assembly code and related optimise methods; to provide a practical suggestion in algorithms for minising overall jumps in compilers.

The remainder of this project report will be structured as follows.

- At first, this paper will describe the problem of jumping cost in compiler and provide a mathematic model for general approach and analysis.
- After introducing the mathematic model, the definition of implicit links would be presented, transformation of the specific problem and potential algorithms followed.
- Then, an outline of implementation is given with testing, benchmarking and optimisation using sample program.
- Finally, practical algorithmic methods and development are discussed.

1.1 Problem description

Ghica's Geometry of synthesis (GoS) compiler (Ghica, 2007) can compile programs to continuation-passing style (CPS-style) "flat" (not procedures) code. However, CPS-style compilation uses a lot of jumping at the level of assembly.

Compared with normal flow control, jumps are more expensive in CPU clock cycles. CPU would spend 10 more times for non-local jumps when cache missing (Austin et al, 2002).

Thus, minimising overall jumps in programs is an effective way in optimising CPS-style compliers.

1.2 Mathematic model

To analyse and proceed the project, an appropriate model is needed to describe the structure of assembly code and relation of jumps. Graph is used to describe and abstract this mathematical problem, which has more theoretical bases to analyse. After the model is set, it is rational to design algorithms to solve the problem, and it is possible to evaluate the algorithm and find out the practical methods.

The structure of assembly code can be defined on a directed graph $\mathrm{G} = (\mathrm{V}, \mathrm{A})$ that jumps are asymmetric.

- The set $V = \{1, \dots \dots, n\}$ is the vertex set which represents the assembly code block set or label set.
- The set $A = \{(i, j): i, j \in V, i \neq j\}$ is a directed edge set that each edge represents the jump from vertex i to vertex j on G.
- Then, a cost matrix $C = (c_{i,j})$ can be defined on A that cost $c_{i,j}$ is defined as the amount of memory size or pages the jump from i to j jumps whether i is adjacent followed by j in assembly code.

Thus, if there are several vertices $\mathrm{E} \subset \mathrm{V}$ between i and j, the formulation of cost is as follow:

$$\mathrm{C_{i,j}} = |\mathrm{c_{i,j}}| + \sum_{\mathrm{k} \in \mathrm{E}} \mathrm{w_k}$$

where:

$\mathrm{w_k}$ means the total amount of memory size or pages in vertex k

To this, the formulation of jumping cost is confirmed in the graphical model. It presents that the cost of jump is strongly related to the set E, which is decided by the sequence of the assembly code blocks.

Therefore, solving the jumping minimising problem in this project is to figure out a sequence of assembly code blocks that the total cost of jumps in this sequence is minimised. Using a list to represent the order of assembly code blocks, the jumping minimising problem hence can be transformed into flattening a graph into a list that makes overall jumping cost minimum.

1.3 general solution

After confirming an appropriate model in jumping minimising problem, it is possible to evaluate the total cost of jumps in specific situations.

Given a sequence of assembly code blocks in the mathematic model mentioned above, to calculate the overall cost of jumps, the cost formulation can be obtained as follow:

$$C_{total} = \sum_{i=1}^{n-1} \sum_{j>i}^{n} C_{i,j}$$

$$C_{total} = \sum_{i=1}^{n-1} \sum_{j>i}^{n} (|c_{i,j}| + \sum_{k \in E_{i,j}} w_k)$$

$$C_{total} = \sum_{i=1}^{n-1} \sum_{j>i}^{n} |c_{i,j}| + \sum_{i=1}^{n-1} \sum_{j>i}^{n} (\sum_{k \in E_{i,j}} w_k)$$

Because the sequence of assembly code is given, all the $\sum_{k \in E_{i,j}} w_k$ term can be pre-calculated based on follow relationship in time $O(n^2)$:

$$\sum_{k \in E_{i,j}} w_k = \sum_{k \in E_{i,j-1}} w_k + w_j$$

Thus, from the formulation above, the time complexity of checking the total jumping cost for a given assembly code sequence is $O(n^2)$.

A general solution that check all possible sequences to figure out the minimum can be designed naturally. However, the permutation of a graph (size of n) has $A_n^n = n!$ probabilities, which makes this algorithm cannot be solved in polynomial time, which is not acceptable for a compiler.

Therefore, brute-force approach is not considerable, further optimisation analysis and methods are needed to proceed this project.

2 Optimisation

This section will present how this project proceed involving in analysis, problem specification and algorithm designing in the jumping minimising problem for compiler based on the mathematic model and general solution above.

2.1 Dynamic programming analysis

As for sequence problem, dynamic programming is a common method approach. The main idea of dynamic programming in sequence problem is breaking a complex problem into a set of sub-problem which is simple to solve then figure out the solution of this complex problem using the results of its sub-problem. When a same sub-problem occurs, the algorithm is not needed to recompute its solution, which can save computation time storage space.

The key of using dynamic programming in jumping minimising problem is how to construct the solution sequence (for graph of n vertices) using the solution sequence of its sub-problem (graph of n-1 vertices and so on).

However, as for the jumping minimising problem in this project, it is hard to figure out a common stategy to construct a solution with minimal jumping cost by using the minimal jumping cost sequence of its sub-graph. In some situation, maximising the jumping cost of sub-graph is needed to minimising the overall jumping cost of the whole graph.

The approximate dynamic programming algorithm, which is described using recursion as follow, can provide a approximate solution sequence in $O(n^3)$.

1. If the size of graph is equal to 1 then return the sequence only with this vertex. Regards this sequence as the minimsed cost sequence of graph with 1 vertex and return.
2. Use recursion to calculate the minimsed cost sequence of graph with n-1 vertices.
3. Iteratively link a vertex from the set of vertices which are not in the the

minimsed cost sequence of graph with n-1 vertices with this sequence.

4. Regards the sequence with minimised cost in iteration as the minimsed cost sequence and return.

In the test, the approximate dynamic programming algorithm based on this approximate evaluation even work worse, which would generate assembly code having more jumping cost than original code.

Therefore, dynamic programming method cannot provide a prefer solution in the jumping minimising problem in this mathematic model. To proceed the jumping minimising, a measure to specify the problem to proceed the project is needed.

2.2 Implicit link

Considering about the rationale of this project, the most vital reason for this jumping minimising problem is to prevent programs from time wasting caused by cache missing in non-local jumps, which would occupy a large amount of clock cycles in program running. Hence, algorithms can be designed by avoiding non-local jumps to minising the possiblility of cache missing in jumping.

Thus, implicit link can be defined by evaluating parameters of jumps in the mathematic model:

- The jump link with adjacent code blocks ($A' = \{(i,j): i,j \in V, |i-j| = 1\}$), this means $\sum_{k \in E_{i,j}} w_k = 0$.
- The memory size or pages of jump is less than a fixed amount K ($C_{i,j} = |c_{i,j}| < K$).
- In some specific situation ($K \leq 1$), the assembly code jump can be deleted that the jumping can be transformed into natural flow.

After defining the implicit link, to optimise the efficiency of program by avoiding cache missing, the jumping minimising problem can be transformed into maximising the amount of implicit links, which means minimising the amount of non-local jumps (the total number of jumps would not change). By evaluation of implicit links, maximising the amount of implicit

links can decrease the possibility of cache missing, which can improve the efficiency of programs.

2.3 Traveling saleman problem

Based on the definition of implicit link mentioned above, the focus of this project is reansformed into design algorithms that maximise the amount of implicit links by routing and scheduling the order of assembly code blocks. This description is quite similar to Traveling Salesman Problem (TSP), find a best sequence to travel though all the vertices (cities) that maximise income (Bektas, 2006), which was classified as NP-complete problem.

Thus, heuristics approach of TSP can be transformed to generate approximate solution in the implicit link maximising prolem, which can give near optimal solution in a reasonable computational time (Bektas, 2006).

Sequence construction algorithms can be designed beased on heuristics of TSP which are described below. These algorithms stop when a solution is found and is believed that there is 10-15% of optimality in the sequence contruction algorithms (Bektas, 2006).

Nearest neighbor (Greedy) heuristic

Nearest neighbor and greedy heuristics (both present a same algorithm in the implicit link maximising problem) are the the most straightforward TSP heuristic (Bektas, 2006). This heuristic is to always link with the vertex with most potential implicit links that can provide an approximate solution sequence in $O(n^2)$.

The steps of this heuristic is given as:

1. Select the first block of assembly code.
2. Find the block of assembly code having maximal number of implicit jumps with it and link.
3. Is there any unlinked code left? If yes, repeat step 2.
4. Link all the assembly code.

Insertion heuristic

Insertion heuristics are also a king of straight forward algorithm by interatively inserting the rest of vertices into the sub-list represented the solution sequence. The time complexity of insertion heuristic approach also is $O(n^2)$ and the steps of an Insertion heuristic are:

1. Select a block of code a given order.
2. Find an edge in the sub-list and insert the code such that the total number of implicit links will be maximal.
3. Repeat step 2 until no more assembly code remain.

The Held-Karp lower bound

The Held-Karp lower bound is a linear time complexity programming relaxation of the integer formulation for TSP problem (Johnson *et al*, 1996), which is a common way to test the performance of other TSP heuristics. Hence the Held-Karp lower bound can be regarded as an evaluation method to test the quality of jumping minimising algorithms in this project.

2.4 Optimisation

After generating the approximate sequence solution by using above construction heuristic, the quality of the solution. Generally, 2-opt and 3-opt exchange improving heuristic is simplest and most effective in solution optimasation depends on the tour generated by the construction heuristic (Bektas, 2006). Based on 2-opt or 3-opt heuristic, meta-heuristic approaches can provide a definite improvement in sequence optimasation such as tabu search or simulated annealing (Johnson & McGeoch, 1997).

2-opt and 3-opt exchange heuristic

The 2-opt/3-opt exchange heuristic is a common algorithm in TSP. This heuristic in implicit link maximising problem could be removing randomly two adjacent edges (a vertex) from the

solution sequence and reconnecting to create a new solution sequence, which 3-opt algorithm works in a similar fashion. The algorithm would improve the solution sequence when a better sequence found and terminate when no further improvement is possible.

The algorithm of 2-opt improvement is presented follow and the time complexity of 2-opt heuristic is $O(n^2)$.

1. Randomly select a block of code in the solution sequence and remove.
2. Iteratively insert this code block into the sequence that improve the solution sequence when better solution found.
3. Mark this code block and go back to setp 1 for next iteration.

In the test, 2-opt exchange heuristic almost cannot find out a better improvement for given test data. The effect of 2-opt exchange heuristic is not remarkable.

k-opt exchange heuristic

2-opt and 3-opt exchang heuristics are special cases of k-opt exchange heuristic. To improve generated sequence solution already made k-opt move can be applied, however, exchang heuristic would spend extremely large computational time when having k>2. According to the test result of 2-opt improvement method, more k-opt exchange heuristic is not recommended though this method can provide optimisation.

Tabu searching

In general, uses 2-opt exchange heuristic for searching would consider about tabu search to avoid getting stuck in local optimums. Tabu search approach in implicit link maximizing problem can improve the solution sequence by maintaining a tabu list containing bad solution in some scale, which can easily avoid local optimums. There are several ways of implementing the tabu list. For more detail paper by (Johnson & McGeoch, 1995) can be referred. However, the vital problem with the tabu search also is running time, which generally

takes $O(n^3)$ (Johnson & McGeoch, 1997), making the search slower than 2-opt improvement method.

As for optimisation of the constructed solution sequence, there is a variety of improvement method such as simulated annealing, genetic algorithm and ant colony optimization (Johnson & McGeoch, 1997). These optimasation method would spend at less $O(n^2)$ computational time for obvious effect.

Therefore, though there is 10-15% of optimality after the solution sequence constructed (Bektas, 2006), plenty of solution sequence optimasation is not recommended for compiler.

3 Implementation and testing

Taking the algorithms designed in previous sections, implementation and optimasation is implemented as it is described. When testing the algorithms, the effect of algorithms and benchmark are the main part of evaluation. The test data in this section is selected and generated for covering kinds of properties would occur in practice widely.

The programs and optimisation was implemented in language C/C++, compiled by g++. The assembly code generated runs under x86 instruction framework.

3.1 implementation

Data type

According to the description of the mathematic model, the first step of implementation is to define types and operations. Considering about the readability and maintenance of the program, the type definitions is implemented in C type struct as follow.

```
struct Node
{
    int id;
    //the amount of memory size or pages in this vertex
    int weight;
    //the total cost from this vertex to vertex i
    int c[n];
    //the number of potential jumps to vertex i
    int w[n];
} node[n];
```

Evaluate function

Evaluate function is the method for calculation the cost using the fomuration mentioned above (1.2). This function plays an important role for evaluating different definition of implicit link and become a key method for adapting mathematic model for different kind of heuristics in algorithms implementation.

The main struct of evaluate function is presented as follow, and it would be changed in adapting different definition of implicit links and heuristic.

```
int evaluate_function(node i, node j)
{
    //for different defination of implicit link
    if(i.c[j.id] <= K * i.w[j.id]) return i.w[j.id];
    else return 0;

    ...

    //for calculating cost of jumps
    int ret = 0;
    for (int k = i.id + 1; k < j.id ; k++) {
        ret += node[k].weight;
    }
    return ret + i.c[j.id];
}
```

refined algorithm

Cost checking

```
int checking()
{
    // pre-calculate the sum of continuous code block weight
    pre_calc_weight();

    int ret = 0;
    for(int i = 0;i < n;i++)
        for(int j = i + 1;j < n;j++)
        {
            // based on the cost formulation
            ret += Math.abs(c[i][j]) + w[i][j];
        }
    return ret;
}
```

Implicit link counting

```
int implicit_link_counting()
{
    // pre-calculate the potiential number of implicit link in given order
    pre_calc_potential_implicit_link();

    int ret = 0;
    for(int i = 0;i < n - 1;i++)
    {
        ret += evaluate_function(node[order[i]],node[order[i+1]])
    }
```

```
    return ret;
}
```

Approximate dynamic programming

```
void approximat_dynamic_programming()
{
    // initialise the sequence for databack
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
        {
            best_of_subproblem[i][j]=-1;
            databack[i][j]=-1;
        }

    // solve with the graph with n = 1
    for(int i=0;i<n;i++)
    {
        dpNode[0][i]=node[i];
        best_of_subproblem[0][i]=0;
    }

    // dynamic programming in O(n^3)
    for(int i=1;i<n;i++)
    {
        for(int j=0;j<n;j++)
        {
            for(int k=0;k<n;k++)
            {
                if(dpNode[i-1][k].mark[j] == 0 &&
                   best_of_subproblem[i-1][k]!= -1 &&
                   (best_of_subproblem[i][j] == -1 ||
                    leftJoint(node[j],dpNode[i-1][k])
                    > best_of_subproblem[i][j] ))
                {
                    databack[i][j] = k;
                    best_of_subproblem[i][j]
                        = leftJoint(node[j] ,
                                    dpNode[i-1][k]) + dp[i-1][k];
                    dpNode[i][j]=merge(node[j],dpNode[i-1][k]);
                }
            }
        }
    }
}
```

Nearest neighbor heuristic

```
void Nearest_neighbor_heuristic()
{
    // initialise the variables for databack the sequence
    bool mark[n];
```

```
    databack[0] = -1;
    memset(mark,0,sizeof(mark));

    int i = 0;
    mark[i]=1;
    int k=1,now=i,tmp=-1,next;
    // iterate n times to link with all code blocks
    while(k++ < n)
    {
        tmp=-1;
        mark[now]=1;
        for(int j=0;j<n;j++)
        {
            // evaluation the potential number of implicit link
            if(mark[j] == 0 && (tmp == -1 ||
               evaluate_function(node[now], node[j]) > tmp))
            {

                tmp = evaluate_function(node[now], node[j]);
                next = j;
            }
        }
        // memorise the solution sequence
        databack[k-1] = next;
        mark[next] = 1;
        now = next;
    }
}
```

Insertion heuristic

```
void insertion_heuristic()
{
    // initialise the variables for databack the sequence
    top=0;
    for(int i=0 ; i < n ; i++)
        databack[i]=-1;

    int tmp=0;
    for(int i=1;i<n;i++)
    {
        int k=top,temp=-1;
        int insert=top;
        do
        {
            int a,b,c;
            if(databack[k] == -1)
            {
                a=node[k].w[i];
                b=0;
                c=0;
            }
            else
```

```
            {
                a=node[k].w[i];
                b=node[i].w[databack[k]];
                c=node[k].w[databack[k]];
            }
            // calulate the overall implicit link after insertion
            if(temp==-1 || tmp +a +b -c > temp)
            {
                // insert place
                insert = k;
                temp = tmp+a+b-c;
            }
            k = databack[k];
        } while (k!=-1);

        // memorise the solution sequence
        databack[i]=databack[insert];
        databack[insert]=i;
        tmp=temp;
    }
    cout<<tmp;
}
```

3.2 Testing

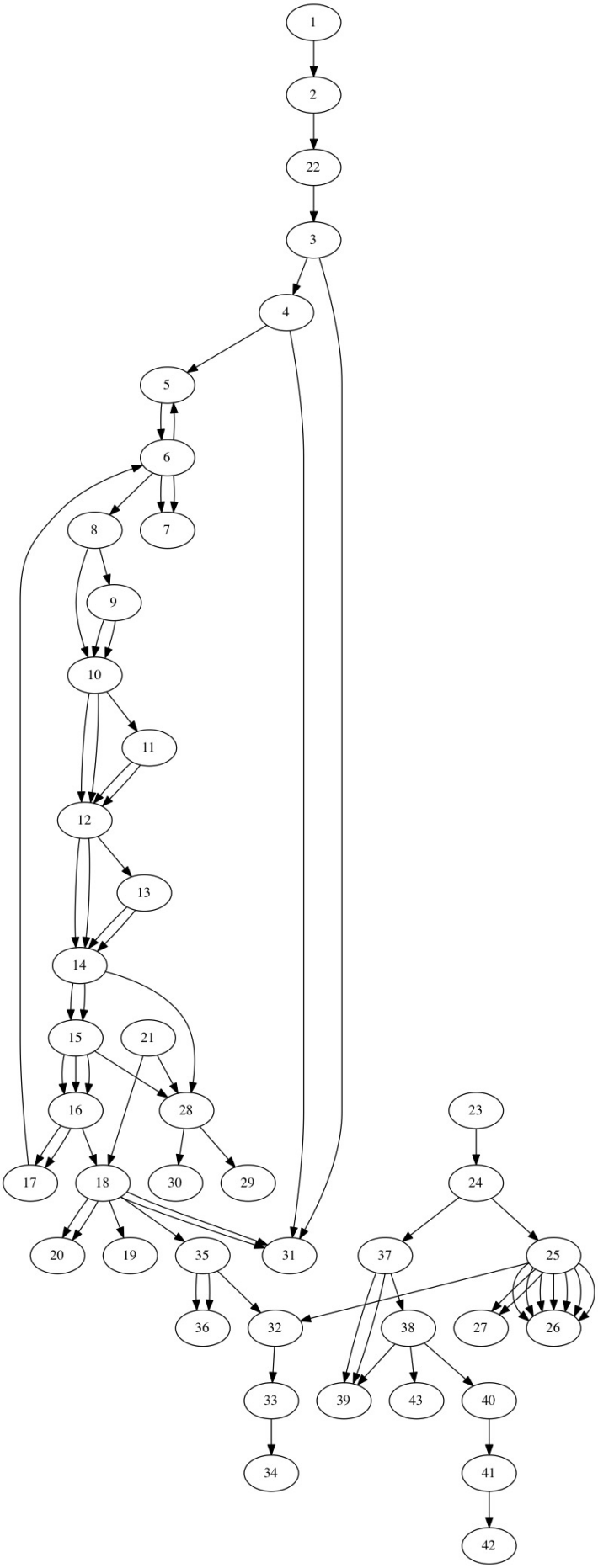
To test the validity of solution, this report use dot language to present the structure of assembly code for giving a clear concept about the algorithms and optimisations. The amount of implicit links is unquestionable become the most vital test metric to evaluate the algorithms.

All the the test environment in this project is Intel Core i5 (2.5 GHz) with 10 GB of memory (1600 MHz DDR3) under OS X operate system (10.11.4).

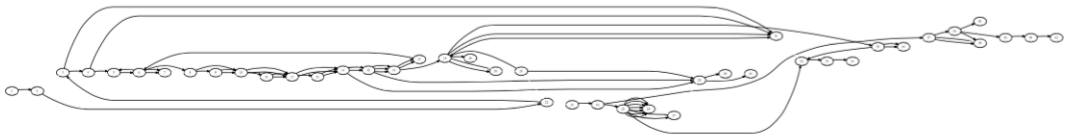
X86 code sample

This report randomly chooses a dump x86 assembly code sample for testing.
http://www.lrdev.com/lr/x86/samp1.html

The structure of this assembly code sample with 42 vetices and 72 jumps is shown as below. Each vertex in this sample represent a function assembly code block. Arrows represent the potential jumping relationship between each code blocks.

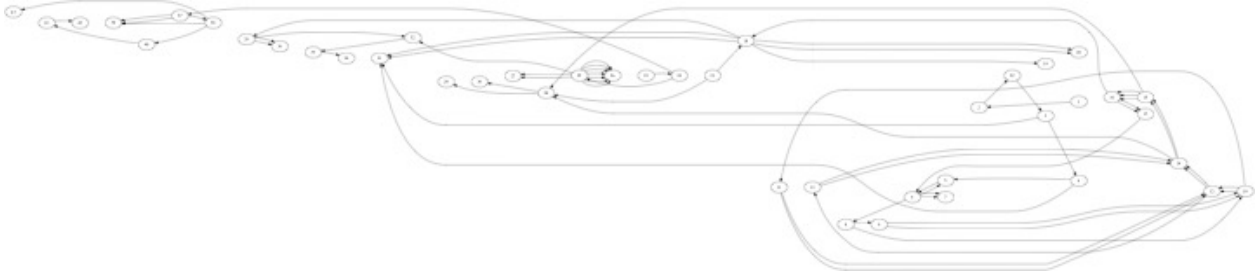

1
2
22
3
4
5
6
8
7
9
10
11
12
13
14
15
21
16
28
17
18
30
29
20
19
35
31
36
32
33
34
23
24
37
25
38
27
26
39
43
40
41
42

The initial assembly code has 42 implicit links with the sequence as follow.

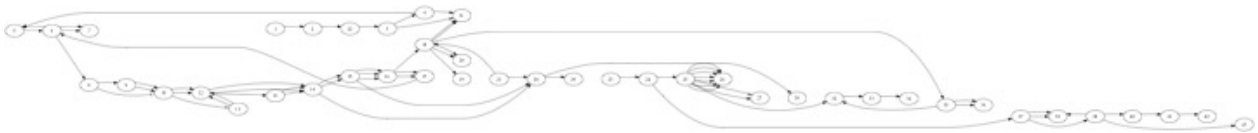


The result of optimization is presented as follow:

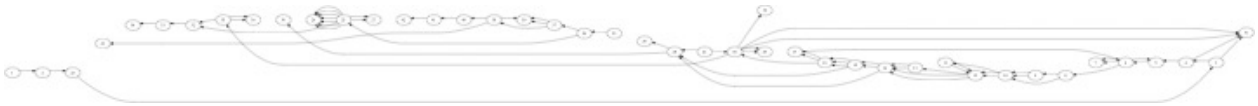
Approximate dynamic programming



Nearest neighbor heuristic



Insertion heuristic



The amount of implicit links after optimization is shown as the table.

Number of implicit links after optimisation		
Approximate dynamic programming	Nearest neighbor heuristic	Insertion heuristic
42	50	52

From the result of optimization above, the approximate dynamic programming algorithm having most time complexity even works worse. There is no effect on the number of implicit links. On the level of mathematic model, it generated assembly code having more jumping cost than original code. Because the insertion heuristic is not so “greedy” compared to nearest neighbor heuristic, the result of insertion heuristic would be better than nearest neighbor heuristic mostly.

To this, benchmarking is necessary to evaluate the effect of jumping minising in this project. More practical suggestion and finding would be found by the process of benchmarking.

3.3 benchmarking

8 queen problem

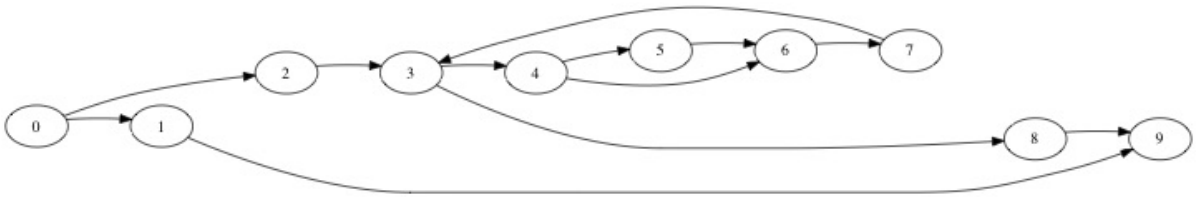
Jumps would occupy more percentage of clock cycles in searching problem compared with other algorithms. Thus, this report uses an assembly code of a common searching problem – 8 queen problem to evaluate the quality of optimisation algorithms for benchmarking.

The assembly code of 8 queen problem is shown as below.

```
      .globl  __Z3DFSi
      .align  4, 0x90
__Z3DFSi:                               ## @_Z3DFSi
      .cfi_startproc
## BB#0:
      pushq   %rbp
Ltmp3:
      .cfi_def_cfa_offset 16
Ltmp4:
      .cfi_offset %rbp, -16
      movq    %rsp, %rbp
Ltmp5:
      .cfi_def_cfa_register %rbp
      subq    $32, %rsp
      movl    %edi, -4(%rbp)
      movl    $1, -8(%rbp)
LBB1_1:                                 ## =>This Inner Loop Header: Depth=1
      cmpl    $9, -8(%rbp)
      jgeLBB1_6
## BB#2:                                ##   in Loop: Header=BB1_1 Depth=1
      movl    -4(%rbp), %edi
      movl    -8(%rbp), %esi
      callq   __Z7Ifblockii
      cmpl    $0, %eax
      je LBB1_4
## BB#3:                                ##   in Loop: Header=BB1_1 Depth=1
      leaq    _a(%rip), %rax
      movslq  -8(%rbp), %rcx
      movslq  -4(%rbp), %rdx
      imulq   $36, %rdx, %rdx
```

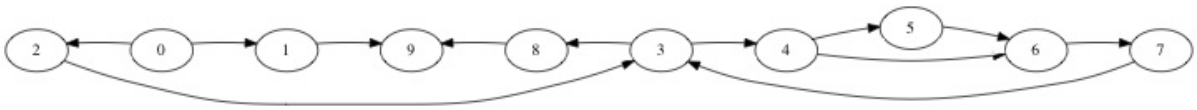
```
      addq    %rdx, %rax
      movl    $1, (%rax,%rcx,4)
      movl    -4(%rbp), %esi
      addl    $1, %esi
      movl    %esi, %edi
      callq   __Z3DFSi
      leaq    _a(%rip), %rcx
      movslq  -8(%rbp), %rdx
      movslq  -4(%rbp), %r8
      imulq   $36, %r8, %r8
      addq    %r8, %rcx
      movl    $0, (%rcx,%rdx,4)
      movl    %eax, -20(%rbp)         ## 4-byte Spill
LBB1_4:                               ##   in Loop: Header=BB1_1 Depth=1
      jmp LBB1_5
LBB1_5:                               ##   in Loop: Header=BB1_1 Depth=1
      movl    -8(%rbp), %eax
      addl    $1, %eax
      movl    %eax, -8(%rbp)
      jmp LBB1_1
LBB1_6:
      movl    $1, %eax
      addq    $32, %rsp
      popq    %rbp
      retq
    .cfi_endproc
```

This report spiltting the assembly code by the labels and the structure of this program is presented as follow.

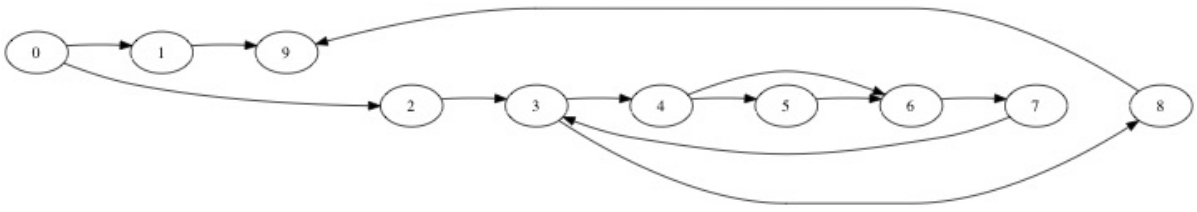


In this section, using insertion heuristic algorithms, optimised versions of assembly code is obtained (under two kind of definition of implicit link).

Version 1



Version 2



To benchmarking the program and distinguish algorithms, the project run 10,000 times of these assembly code in each test cycles. The record of runtime is presented in the table.

Runtime record of 8 queen problem optimisation

Origin	Version 1	Version 2
real 0m3.998s user 0m3.939s sys 0m0.027s	real 0m4.289s user 0m4.235s sys 0m0.024s	real 0m3.997s user 0m3.959s sys 0m0.019s

From the the structure of the assembly code, version 1 seems to have most implicit links but having most “backward” arrows and most computational time. Therefore, backward jumps cannot be defined as implicit link which could have high possibility in causing cache missing effect. Origin version and version 2 have a same number in implicit link that they runtime is similar.

To this, delete the jumps of implicit links that transform the jumps into natural flow in this aseembly part and generate a new version program of version 2. The runtime decrease to approximate 3.93s, which is an obvious effect compared with other programs having same number (7) of implicit links.

Therefore, deleting the jumping code of implicit links have a great effect on jumping minimising problem in this project.

more tppical struct of assembly code needed to benchmark

4 Development and conclusion

avoid backward jumps – Topological Sorting

Bibliography

Austin, T., Larson, E., & Ernst, D. (2002). SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2), 59-67.

Bektas, T. (2006). The multiple traveling salesman problem: an overview of formulations and solution procedures. *Omega*, 34(3), 209-219.

Bonyadi, M. R., Shah-Hosseini, H., & Azghadi, M. R. (2008). *Population-based optimization algorithms for solving the travelling salesman problem*. INTECH Open Access Publisher.

Ghica, D. R. (2007). Geometry of synthesis: a structured approach to VLSI design. *ACM SIGPLAN Notices*, 42(1), 363-375.

Johnson, D. S., & McGeoch, L. A. (1997). The traveling salesman problem: A case study in local optimization. *Local search in combinatorial optimization*, 1, 215-310.

Johnson, D.S; McGeoch, L.A. & Rothberg E.E (1996). Asymptotic Experimental Analysis for the Held- Karp Traveling Salesman Boun. *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, 341-350.