Minimising overall jumps in CPS-style compilers

TIANYU LI
(supervisor: Dan Ghica)

OUTLINE

1.1 DESCRIPTION

- Ghica's *Geometry of Synthesis* (Ghica, 2007) compiler can compile to CPS-style (continuation-passing style) "flat" (not procedures) code
- CPS-style compilation uses a lot of jumping at the level of assembly
- Non-local jumps are expensive
  - Normal flow of control is cheapest
  - Local jumps are more expensive
  - Non-local jumps are very expensive (cache miss)

Minimise jumping costs!

1.2 GRAPH

Directed graph $G = (V, A)$

$V = \{1, \dots \dots, n\}$ – vertex set represents code blocks or labels

$A = \{(i, j): i, j \in V, i \neq j\}$ – directed edge set represents jumps

Cost matrix $C = (c_{i,j})$

- can be defined on A that cost $c_{i,j}$
- the amount of memory or pages the jump from i to j jumps

flattening a graph into a list

1.3 GENERAL SOLUTION

the formulation of cost:

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

Where $\sum_{k \in E} w_k$:

$\quad$ *the total amount of memory or pages between i , j*

General solution

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

However, we cannot consider a brute-force approach.

($A_n^n = n!$ probabilities for sequence problem)

1.3 Dynamic Programming

Dynamic programming
(also known as dynamic optimisation) is an efficient method to optimise permutation problems commonly.

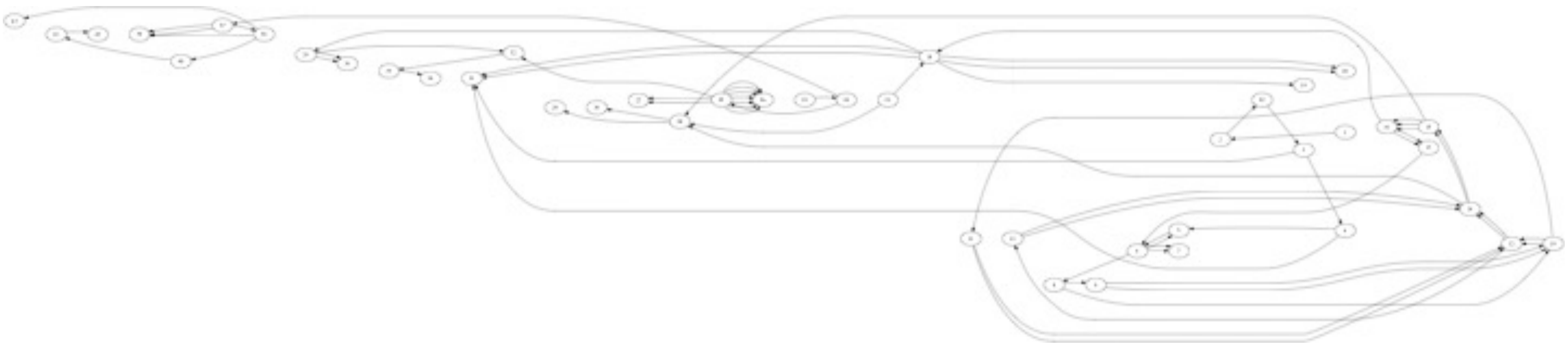
- Figure out the solution of this complex problem using the results of its sub-problems.
- When a same sub-problem occurs, the algorithm does not need to recompute the solutions.

KEY:
construct the solution sequence (for graph of n vertices) using the solution sequence of its sub-problems (graph of n-1 vertices and so on).

Same strategy – might be a NP complete problem

1.3 Approximate Dynamic Programming

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 iterations as the minimsed cost sequence and return.



Not recommended

2.1 DEFINATION

Direction - avoid cache misses

Implicit link:

1. The jumps between adjacency blocks ($\sum_{k \in E_{i,j}} w_k = 0$)
2. The cost of jump is less than K ($C_{i,j} = |c_{i,j}| < K$)
3. The jumps can be deleted in some situation ($K \leq 1$)

The jumping minimising problem was transformed into maximising the amount of implicit links for assembly code.

Called implicit link maximising problem

2.2 Traveling salesman problem (TSP)

Implicit link maximising problem:

- Looking for a list
- To link with all the blocks of assembly code
- Maximising the amount of implicit links

Traveling salesman problem (Bonyadi *et al*, 2008):

- Looking for a route
- To travel through all the cities
- Maximising the value salesman gets (minimising the length)

2.2 TSP

Approximate approaches (Bektas, 2006) :

Nearest neighbor (Greedy) heuristic

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

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 code remain.

2.3 Sequence Improvement

Commonly, 2-opt and 3-opt exchange heuristic is applied for improving the solution.

- 2-opt algorithm removes randomly two links from the already generated list, and reconnects two new links.
- 3-opt works in a similar fashion.

Other ways of improving the solution is to apply meta-heuristic approaches such as tabu search or simulated annealing using 2-opt and 3-opt (Johnson & McGeoch, 1997).

2.3 2-opt exchange

- Time complexity: $O(n^2)$

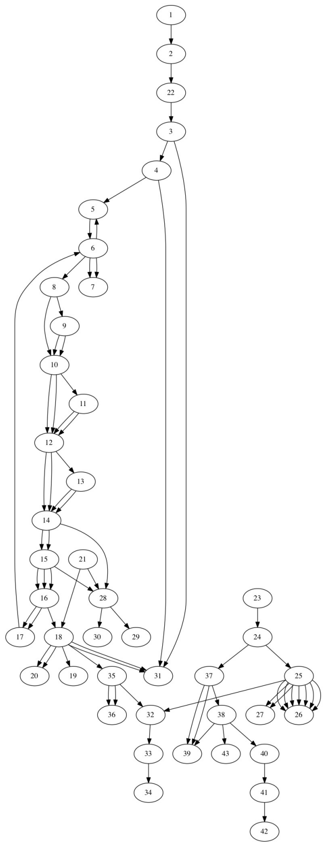
1. Randomly select a block of code in the solution sequence and remove.
2. Iteratively insert this code block into the sequence that improve the solution sequence when better solution found.
3. Mark this code block and go back to step 1 for next iteration.

More time, more efficient!

3.1 TESTING

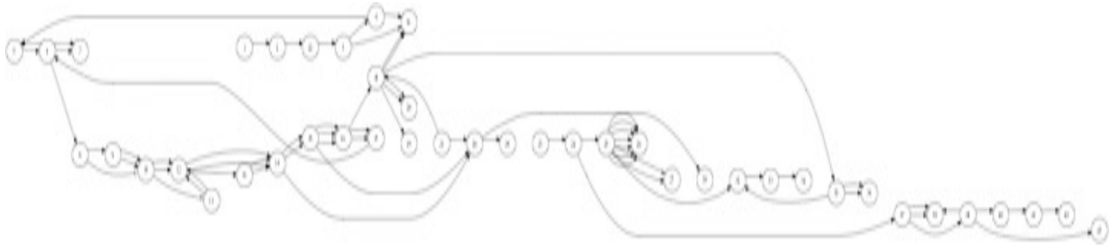
SAMPLE:
http://www.lrdev.com/lr/x86/samp1.html

x86 assembly code:
number of implicit links:42

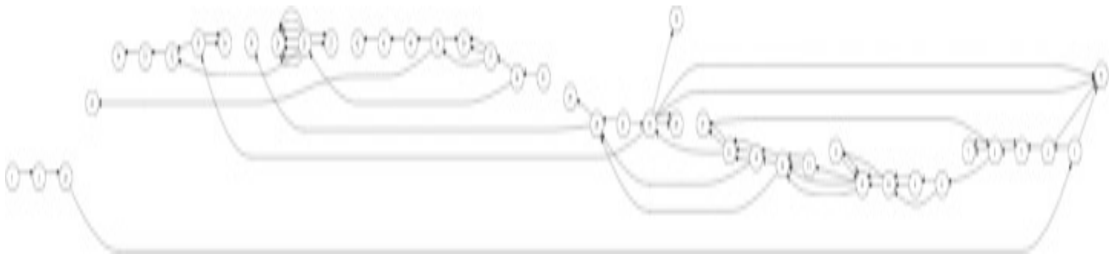


A vertex represents a function

Nearest neighbor (Greedy) heuristic:
number of implicit links:50



Insertion heuristic
number of implicit links:52



Insertion is not so "greedy"

3.2 BENCHMARKING

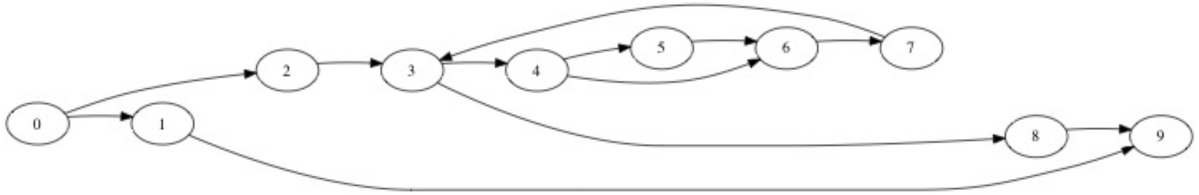
10,000 times of 8 queens problem

Runtime record of 8 queen problem optimisation

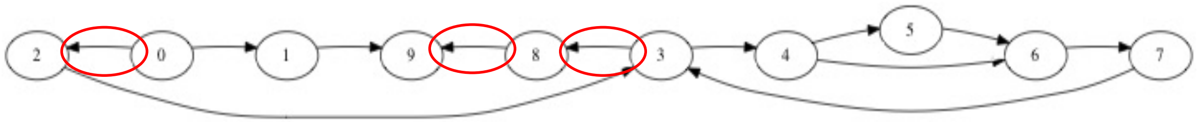
Origin	Insertion	Nearest neighbor
real 0m3.998s user 0m3.939s sys 0m0.027s	real 0m4.289s user 0m4.235s sys 0m0.024s	real 0m3.997s user 0m3.959s sys 0m0.019s
Implicit links: 7	Implicit links: 9	Implicit links: 7

3.2 8 QUEENS PROBLEM

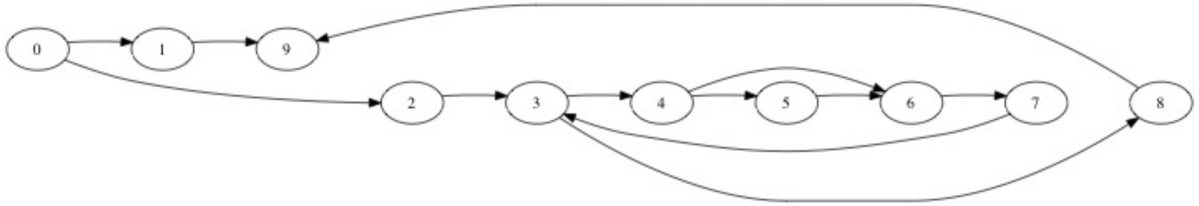
Origin



Insertion



Nearest neighbor



A vertex represents a label (a jump)

Backward jumps cannot be seen as implicit link

3.2 BENCHMARKING

100,000 times of 10,000 random jumps

Runtime record of random generated code

Random_generated	ordered	implicit_link_deleted
real 0m9.948s user 0m9.917s sys 0m0.015s	real 0m4.637s user 0m4.620s sys 0m0.011s	real 0m3.055s user 0m3.041s sys 0m0.008s
	-53%	-34%

3.3 FINDINGS

1. Maximising implicit links can accelerate the program.
2. Normally, there are only conditional jumps and force jumps (potential less than two branched)
3. The way of splitting code makes different
4. Avoid backward jumps
5. Delete the jumps for implicit link

4

CONLUSION

1. Mathematic model
   – General solution
   – Approximate dynamic programming

2. Define the implicit link
   – Generalized cost function
   – Avoid cache missing

3. Algorithms from TSP
   – heuristics and improvement

4. Evaluation and testing

REFERENCES

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.

THANK YOU FOR YOUR ATTENTION