EECS 527 - Project 1

Topic

Implementing Hypergraphs, Basic Algorithms for Linear Placement

It is recommended that you finish Homework 1
before starting this assignment.

The size of tar.gz should be <100kb,
do not include binaries or large benchmarks.
All your source codes should be there,
with a Makefile and a README file (see below).

Technical requirements

Your programs must be written in C or C++
and compile with the g++ compiler on Linux and Solaris.
You are encouraged to use C++ and STL,
but this is not required for Project 1.

The following versions of g++ are accepted:
g++ 3.0-3.3
The version of g++ that you used should be mentioned
in the README file and the Makefile.

Your programs must be able to produce "reasonable" vertex orderings
for all 18 ibm benchmarks on Linux and Solaris
workstations with no more than 256Mb RAM.
"Reasonable" is defined as
"not too far from the second best submission
by students in this course". See below for optimization objectives.
The resulting vertex ordering should be saved to the file output.txt,
one integer per line.

Algorithms and Optimization Objectives

Your programs must minimize the total span (wirelength)
of hypergedges, and in a separate category,
minimize max-cut. You must use the same IBM benchmarks
as in Homework 1, but can ignore vertex weights.

Both optimizations in this Project are unconstrained.

Your programs will be judged based on whether they run well,
based on their runtime and solution quality.

In terms of algorithms, you must implement
trivial "dart-throwing" --- order vertices
at random, evaluate the objective function,
repeat N times and pick the best ordering.

If one is trying to implement more intelligent algorithms
to minimize max-cut and total span, it is worth realizing
that vertices connected by hyperedges should typically be
fairly close to each other in a good vertex ordering.
Following this reasoning, you are asked to try vertex orderings
produced by the Breadth-First Search (BFS)
and the Depth-First Search (DFS) --- graph traversals
that visit vertices by following hyper-edges.

Implement BFS and DFS, perhaps using stack<> and queue<> from STL.
Do they ever/typically/always produce better results than dart-throwing?
Come up with good ways of selecting the starting vertex in BFS/DFS,
and feel free to implement various tie-breaking schemes.
You may also run, say, BFS with two
different starting points and then choose the better
of the two. Think of other variants along these lines.

Documentation

The README.txt file (or README.ps or README.pdf)
enclosed with your implementation must explain how
to run your program to optimize max-cut and total span.
Explain how to request the dart-throwing algorithm, BFS, DFS, etc.
Say whether a well-implemented bare-bones DFS
consistently runs faster than a well-implemented bare-bones BFS,
substantiate this by empirical data and explain why.
The README.* file must explain algorithmic improvements
to BFS and DFS that you implemented and motivate them.
Give empirical data (runtime, total span, max-cut and
memory usage) for all of your implementations
on the 18 IBM benchmarks and compare them to results of dart-throwing.
Rank the 18 benchmarks in the order of difficulty.

Optional: For 10% extra credit, implement any algorithm that does not
contain BFS and DFS that either finds better orderings in

comparable (or better) time OR gives non-trivial improvements
if applied after BFS and, independently, after DFS.
In the latter case, limit the runtime to 100 times
the runtime of BFS.

==============================
ADDITIONAL REFERENCES

Online Makefile tutorials
  Simple intros
    http://oucsace.cs.ohiou.edu/~bhumphre/makefile.html
    http://www.cs.indiana.edu/classes/c304/Makefiles.html
  More depth (may be too much depth for this Project)
    http://www.eng.hawaii.edu/Tutor/Make/
    http://www.gnu.org/manual/make/html_chapter/make_toc.html