A Simple Algorithm for Dominating Set
A **dominating set** of a graph $G$ is a subset $D$ of the vertices of $G$ such that every vertex $v$ of $G$ is either in the set $D$ or $v$ has at least one neighbour that is in $D$. 
Vizing's conjecture concerns a relation between the domination number and the cartesian product of graphs. This conjecture was first stated by Vadim G. Vizing (1968), and states that, if $\gamma(G)$ denotes the minimum number of vertices in a dominating set for $G$, then $\gamma(G \square H) \geq \gamma(G)\gamma(H)$.

Conjecture predicts $\geq 1$ for this graph so it is not tight.

http://en.wikipedia.org/wiki/Vizing's_conjecture
A Map of the Town of Iceberg
The goal of your course project for CSC 425/520 is to develop and test practical algorithms for finding a minimum dominating set for a graph.

The deliverables for the course project include code that should be programmed in C or C++ (Java is too slow for research purposes), a research paper prepared using LaTeX, a web page describing some test graphs, and some slides for a research presentation. The materials developed are intended to be cumulative (each week you will add material to what is already present).

Students will present a summary of their research using the slides created in the last week of classes.

Students in CSC 520 are responsible for one extra algorithm; either a heuristic algorithm, or an algorithm specialized to certain classes of graphs.
Adjacency list:
Adjacency matrix:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Input as:
5
2 1 3
4 0 3 4 2
2 1 4
3 0 4 1
3 1 3 2
A simple but reasonable fast dominating set algorithm (you can implement this for milestone 1):

Data structures:
The graph:

\( n \) = number of vertices
\( A[0..(n-1)][0..(n-1)] \) = adjacency matrix, but I changed the diagonal so that the values are all 1's (because a vertex dominates itself).

\( \text{DELTA} \) = maximum degree of a vertex \( v \)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Each vertex has a status:

white: not decided

blue: excluded from dominating set

red: included in dominating set

I did not actually record these explicitly although it could be useful in algorithm variants.
To record a partial dominating set:

size
dom[0..(n-1)]

or the minimum dominating set found so far:

min_size
min_dom[0..(n-1)]
\text{n\_dominated}= \text{number of dominated vertices.}

For each vertex \text{v}:
\text{num\_dominated}[\text{v}]= \text{number of times it is dominated by a red vertex.}

\text{num\_choice}[\text{v}]= \text{number of times it could be dominated if all white vertices were red ones. If \text{num\_choice}[\text{v}] is 0 for some vertex, we can back up (solution cannot be completed to a dominating set).}
When programming recursive algorithms, it helps in debugging to have a variable level representing the level of recursion.

Initial call:

```
min_dom_set(0, ...)
```

Declaration of function:
```
int min_dom_set(int level, ...)
```

Recursive calls:
```
min_dom_set(level+1, ...)
```
At the top:

#include <stdio.h>

#include <stdlib.h>

#define NMAX 500

#define DEBUG 1
Debugging:

#if DEBUG
    printf("Level %3d: ", level);
    print_vector(size, dom);
    printf("Number of vertices dominated: %3d\n", n_dominated);
    printf("Number of choices per vertex: \n");
    print_vector(n, num_choice);
    printf("Number of times dominated: \n");
    print_vector(n, num_dominated);
#endif
At a given level, I decide the status of vertex number \textit{level}.

I first try making it \textit{blue} and then \textit{red}. Before returning: change it back to white.
At level 0 we initialize the data structures first:

Implicit: all vertices are white.
\( n_{\text{dominated}} = 0 \)
\( \text{num}\_\text{choice}[v] = \) degree of \( v \) + 1
\( \text{num}\_\text{dominated}[v] = 0 \)

\( \text{size} = 0 \)
\( \text{dom}[i] = \) no values assigned since size is 0
\( \text{min}\_\text{size} = n \)
\( \text{min}\_\text{dom}[i] = i \) for \( i = 0 \) to \( n-1 \)
Tests used to check if we should backtrack (not completable to a dominating set smaller than the min so far).

If for any vertex $v$, $\text{num\_choice}[v]$ is 0 then return.

Set $n\_extra = \left\lfloor \frac{u}{\Delta + 1} \right\rfloor$

$u =$ number of undominated vertices

$\Delta =$ maximum degree of a vertex

If $\text{size} + n\_extra \geq \text{min}$ then return.
Termination condition:

At level n (all vertices v have a status, and \text{num\_choice}[v] is at least 1 so dominated) or if all vertices are dominated:

If size < min_size
  copy the current dominating
  set \text{dom} to \text{min\_dom}
  and set min_size = size.
End if

return
The exhaustive backtrack:

Set $u = \text{level}$.

Try vertex $u$ as blue (excluded from dominating set):

For each neighbour $v$ of $u$ as recorded in $A$, decrement $\text{num\_choice}[v]$.

Call the routine recursively.

For each neighbour $v$ of $u$ as recorded in $A$, increment $\text{num\_choice}[v]$.

Recursive routines should restore data structures to avoid need to copy them.
Try vertex $u$ as red (in dominating set):

Add $u$ to $\text{dom}$.
For each neighbour $v$ of $u$ as recorded in $A$, increment $\text{num\_dominated}[v]$.
Update $\text{n\_dominated}$.

Call the routine recursively.

Restore data structures and return.

Recursive routines should restore data structures to avoid need to copy them.
Level 0: initially all vertices are white:
Levels 0, 1, 2: vertices are set to blue initially. Level 3: try blue and then 0 has num_choice[0]=0 so back up from level 4 then color 3 red.
Levels 4, 5 choose blue initially, level 6 tries blue (but then num_choice[1]=0 at level 7) and then red.
Levels 7, 8 try blue initially. Level 9 try blue (but then num_choice[2]= 0 at level 10) and then red.
Record this (min_size, min_dom) since better than best so far (10) and return.
At level 8, try vertex 8 as red. At level 9: size=3, n_extra= \[\lceil \frac{2}{(3+1)} \rceil \] = 1
size + n_extra = 4 ≥ 3 = min_size so return.
At level 7, try vertex 7 as red. At level 8:
size=3, n_extra= \lceil 2/(3+1) \rceil =1
size + n_extra = 4 ≥ 3 = min_size so return.
At level 5, try vertex 5 as red. At level 6: size=2, n_extra= ⌈4/(3+1)⌉ =1
size + n_extra = 3 ≥ 3 = min_size so return.
At level 4, try vertex 4 as red. At level 5: 
size=2, n_extra= ⌈3/(3+1)⌉ =1 
size + n_extra = 3 ≥ 3 = min_size so return.
At level 42 try vertex 2 as red. At level 3: size=1, n_extra= ⌈6/(3+1)⌉ = 2
size + n_extra = 3 ≥ 3 = min_size so return.

Similarly, we return at levels 1 and 0 after trying red.
Fullerenes

- Correspond to 3-regular planar graphs.
- All faces are size 5 or 6.
- Euler’s formula: exactly 12 pentagons.
Command file for running on small fullerenes (run_com):

time a.out 1 < c020 > o020

time a.out 1 < c024 > o024

time a.out 1 < c026 > o026

time a.out 1 < c028 > o028

time a.out 1 < c030 > o030

time a.out 1 < c032 > o032

time a.out 1 < c034 > o034

time a.out 1 < c036 > o036

time a.out 1 < c038 > o038

time a.out 1 < c040 > o040

time a.out 1 < c042 > o042

time a.out 1 < c044 > o044

time a.out 1 < c046 > o046

time a.out 1 < c048 > o048

time a.out 1 < c050 > o050

time a.out 1 < c052 > o052

time a.out 1 < c054 > o054

time a.out 1 < c056 > o056

time a.out 1 < c058 > o058

time a.out 1 < c060 > o060

To run this:
source run_com
Timing data for all small fullerenes:

<table>
<thead>
<tr>
<th>n</th>
<th>#</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>0.004</td>
</tr>
<tr>
<td>28</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>3</td>
<td>0.004</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
<td>0.02</td>
</tr>
<tr>
<td>34</td>
<td>6</td>
<td>0.016</td>
</tr>
<tr>
<td>36</td>
<td>15</td>
<td>0.076</td>
</tr>
<tr>
<td>38</td>
<td>17</td>
<td>0.092</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
<td>0.672</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>lb</th>
<th>#</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>11</td>
<td>45</td>
<td>0.504</td>
</tr>
<tr>
<td>44</td>
<td>11</td>
<td>89</td>
<td>2.6</td>
</tr>
<tr>
<td>46</td>
<td>12</td>
<td>116</td>
<td>2.728</td>
</tr>
<tr>
<td>48</td>
<td>12</td>
<td>299</td>
<td>13.66</td>
</tr>
<tr>
<td>50</td>
<td>13</td>
<td>271</td>
<td>13.592</td>
</tr>
<tr>
<td>52</td>
<td>13</td>
<td>437</td>
<td>58.023</td>
</tr>
<tr>
<td>54</td>
<td>14</td>
<td>580</td>
<td>58.44</td>
</tr>
<tr>
<td>56</td>
<td>14</td>
<td>924</td>
<td>295.042</td>
</tr>
<tr>
<td>58</td>
<td>15</td>
<td>1205</td>
<td>248.143</td>
</tr>
<tr>
<td>60</td>
<td>15</td>
<td>1812</td>
<td>1109.341</td>
</tr>
</tbody>
</table>

For 40: 0.672u 0.000s 0:00.67 100.0% 0+0k 0+24io 0pf+0w

= 4.9 minutes

= 18.5 minutes
Only fullerene isomer \( C_{56:649} \) has dominating set order 14.