CSCE 420
Programing Assignment #6
due: Tues, Nov 11 (by start of class)

Objective

The goal of this assignment is to apply the backtracking algorithm you have written in C++ for solving Constraint Satisfaction Problems to solving a **multi-agent coordination** problem.

Background

The problem involves agents at starting nodes on a 3x3 grid who each want to get to a different destination. However, they cannot occupy the same node at the same time, nor use the same edge at the same time. Time is assumed to be discrete. Thus, the agents must plan routes to get to their destinations that are mutually compatible, which may require move around each other, or even pausing till another agent passes. We want to solve this coordination problem in a *centralized* way, using one algorithm to plan all the routes simultaneously (as opposed to a distributed approach, where the agents each make their own decisions and have to negotiate/cooperate).

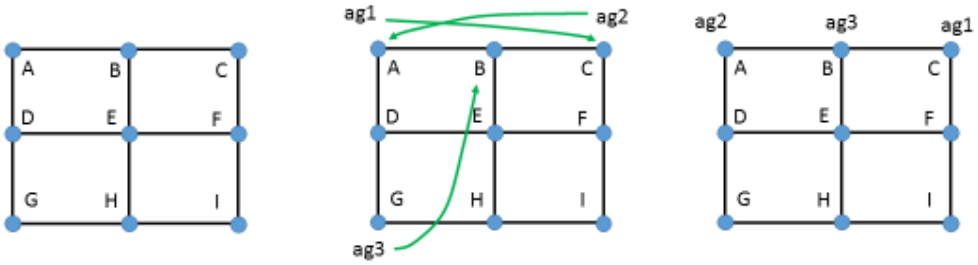
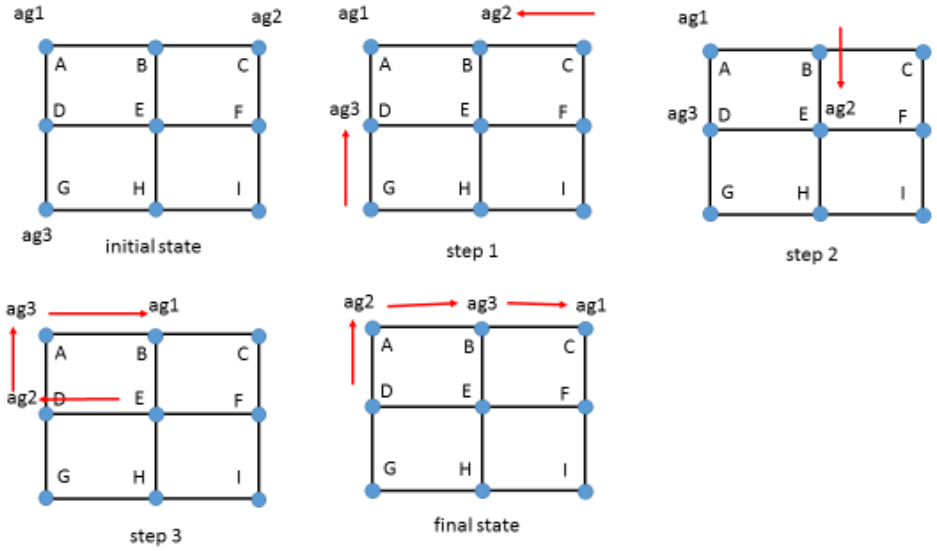


Figure 1: a) a 3x3 interconnect. b) the initial locations of 3 agents and, c) their desired destinations.

In the example shown in Figure 1b, there are 3 agents, two of which want to exchange places (ag1 and ag2, to swap between nodes A and C), and the third to go between them (ag3, to go from to node G to B). This problem can be solved in 4 steps, as shown below:



Finding mutually consistent paths gets harder with more agents in this constrained environment. Of course, there are limits as to what can be achieved on a 3x3 interconnect. For example, if you start with 9 agents, one at each node, there is no room for them to move around each other and they would be locked in place. But you should design your program to handle different combinations of source/destination requests from different numbers of agents.

Applications

There are numerous problems in the real world that feature this kind of constrained coordination among multiple agents. For example, consider **air traffic control**. In a given sector of airspace, the (centralized) air traffic controller must plan routes for multiple aircraft each headed in different directions, directing them to make small changes in the flight trajectories as necessary to maintain adequate separation. Another analogy would be to coordinating **trains** in a transportation network, where the nodes are train stations, the edges are tracks, and the goal is to plan a sequence of moves so that all trains get safely from their source to their destination without collisions. Finally, this problem can also be applied to data traffic on a **computer interconnect**. Suppose the nodes represent connection points among 8 cores or CPUs in a server, and the edges in the 3x3 grid represent **data buses** between them (the nodes might be latches or buffers). The goal would be to find a sequence of data transfers such that the integrity of each signal/message would be maintained and no collisions/corruption would occur.

Modeling this Coordination as a CSP

There are many ways to solve this problem. We will be modeling it as a CSP in this assignment. How can we express the constraints of this problem in such a way that solving the CSP is equivalent to finding non-intersecting paths for the agents? Recall that defining a CSP requires defining: 1) variables, 2) domains, and 3) constraints. As variables, we will create hybrid names that combine the name of the agents with a time index. We will assume that the problem can be solved in a finite number of steps (like 4, or 5, or 6), which should be passed into the program as an input argument. Thus, if the agents are named agX and agY, and the max time is 4 steps, then there will be 8 CSP variables to construct: agX-1, agX-2, agX-3, agX-4, agY-1, agY-2, agY-3, agY-4. That is, one variable for each agent and each time point. The domains will then be the list of nodes, {A,B...I}, giving 9 possible locations for each agent at each time step. Finally, the constraints should capture things like: 1) an agent must be at exactly one node at each time step, 2) 2 agents can't be at the same node at the same time, and 3) an agent must be at the same or adjacent node at successive time points. Of course, the domains for the first and last time points can be singletons, since the starting and destination nodes of each agent are specified by the problem instance.

Implementation

You will use the same CSP code you wrote in C++ for the previous project (map-coloring). This problem is hard enough that you will definitely need the MRV heuristic.

The focus of this assignment is more on coming up with the right encoding of this problem as a CSP, as described above. You will probably find it useful to write a program to **generate** the problem specification file for different problem instances. For example, for the 3-agent problem above, the

problem instance might be described in an input file like this, which describes the agents, and their source and desired destination nodes. (You can assume no agents will start or end on the same node.)

```
# filename "destinations.3-agents"
ag1 A C
ag2 C A
ag3 G B
```

To generate the constraints, you also have to know about the graph connectivity. A file describing the interconnect graph should be read in, that defines the nodes and edges (will be made available to you on the course website to download). The first line contains the number of nodes and the number of edges. This is followed by a row for each node containing id and coordinates (not used), and then a row for each edge, giving pairs of node id's that are connected.

```
# filename "graph.interconnect3"
# there are 9 nodes and 12 edges
9 12
A 1 1
B 1 2
C 1 3
...
A B
B C
D E
...
```

Your pre-processing program should read these files in and generate the specification file, which contains the variables, domains, and constraints (similar to the input for the Australia problem last time). An important parameter in this process is the time limit T for the agents (i.e. max number of steps or moves to get to their destinations). The number of steps affects the variables that get created, and the constraints that relate them. Some coordination problems might not be solvable in 4 steps. But you can increase it to 5 or 6 to see if there is a solution. For uniformity, assume time 1 is the initial state and time T is the final state.

```
> generate_CSP_spec destinations.3-agents graph.interconnect3 5
# variables
ag1-1 ag1-2 ...
# domains
ag1-1 A
ag1-2 A B C D E F G H I
...
# constraints
...
```

Similar to last time, some of your constraints might be inequality constraints. For example, one constraint might be "neq ag1-2 ag2-2", which could be interpreted as the location of ag1 at time 2 cannot be the same as the location of ag2 at the same time. However, you will probably find it necessary to use other types of constraints as well. For example, you might find *equality* constraints to

be useful. Or some constraints might compare and test the consistency of more than 2 variables at a time. Whatever constraints you generate in your CSP specification file, your CSP program should be able to read them in, and you will have to extend your satsified() function to evaluate them.

What to Turn in

- You will submit your code for testing using the web-based CSCE *turnin* facility, which is described here: https://wiki.cse.tamu.edu/index.php/Turning_in_Assignments_on_CSNet
- Include a brief document that describes how to compile and run your code
- Include an example of the output, showing the sequence of coordinated moves (similar to the transcript below). You might want to show this for a simpler 2-agent version of the problem, the 3-agent problem instance described above, and a representative harder one that you can solve. For each of these, make sure the output shows the **total count of the iterations** made during the backtracking search (either with or without the MRV heuristic), along with the final solution.

Example Run

the input args are: <file describing graph> <file describing agents and src/dest nodes> <time bound>
(mine happens to be implemented in Python, but you should implement yours in C++)
(the 'counts' below show size of domains of each variable at each iteration, which is used by MRV)

```
> generate_CSP_spec graph.interconnect3 destinations.3-agents 5 > CSP1.dat
> python CSP.py CSP1.dat
iter 1
assignment: {}
counts: [(1, 'ac1-1'), (1, 'ac1-5'), (1, 'ac2-1'), (1, 'ac2-5'), (1, 'ac3-1'),
(1, 'ac3-5'), (9, 'ac1-2'), (9, 'ac1-3'), (9, 'ac1-4'), (9, 'ac2-2'), (9, 'ac2-
3'), (9, 'ac2-4'), (9, 'ac3-2'), (9, 'ac3-3'), (9, 'ac3-4')]
next var: ac1-1
trying A

iter 2
assignment: {'ac1-1': 'A'}
counts: [(1, 'ac1-5'), (1, 'ac2-1'), (1, 'ac2-5'), (1, 'ac3-1'), (1, 'ac3-5'),
(3, 'ac1-2'), (9, 'ac1-3'), (9, 'ac1-4'), (9, 'ac2-2'), (9, 'ac2-3'), (9, 'ac2-
4'), (9, 'ac3-2'), (9, 'ac3-3'), (9, 'ac3-4')]
next var: ac1-5
trying C

iter 3
assignment: {'ac1-5': 'C', 'ac1-1': 'A'}
counts: [(1, 'ac2-1'), (1, 'ac2-5'), (1, 'ac3-1'), (1, 'ac3-5'), (3, 'ac1-2'),
(3, 'ac1-4'), (9, 'ac1-3'), (9, 'ac2-2'), (9, 'ac2-3'), (9, 'ac2-4'), (9, 'ac3-
2'), (9, 'ac3-3'), (9, 'ac3-4')]
next var: ac2-1
trying C

...

iter 12
assignment: {'ac2-4': 'A', 'ac2-5': 'A', 'ac2-1': 'C', 'ac2-3': 'D', 'ac3-1': '
G', 'ac3-5': 'B', 'ac1-5': 'C', 'ac1-4': 'B', 'ac1-3': 'A', 'ac1-2': 'A', 'ac1-
1': 'A'}
counts: [(0, 'ac2-2'), (1, 'ac3-4'), (3, 'ac3-2'), (7, 'ac3-3')]
next: ac2-2
trying A False <- this means a constraint was violated
trying B False
trying C False
trying D False
trying E False
trying F False
trying G False
trying H False
trying I False
backtrack from: {'ac2-4': 'A', 'ac2-5': 'A', 'ac2-1': 'C', 'ac2-3': 'D', 'ac3-1
': 'G', 'ac3-5': 'B', 'ac1-5': 'C', 'ac1-4': 'B', 'ac1-3': 'A', 'ac1-2': 'A', '
ac1-1': 'A'}

...

iter 18
assignment: {'ac2-4': 'D', 'ac2-5': 'A', 'ac3-2': 'D', 'ac2-1': 'C', 'ac2-2': '
B', 'ac2-3': 'E', 'ac3-1': 'G', 'ac3-5': 'B', 'ac1-5': 'C', 'ac1-4': 'B', 'ac1-
3': 'A', 'ac1-2': 'A', 'ac1-1': 'A', 'ac3-4': 'A', 'ac3-3': 'D'}
```

```
solved!
solution:
{'ac2-4': 'D', 'ac2-5': 'A', 'ac3-2': 'D', 'ac2-1': 'C', 'ac2-2': 'B', 'ac2-3':
 'E', 'ac3-1': 'G', 'ac3-5': 'B', 'ac1-5': 'C', 'ac1-4': 'B', 'ac1-3': 'A', 'ac
1-2': 'A', 'ac1-1': 'A', 'ac3-4': 'A', 'ac3-3': 'D'}

variable bindings:
ac1-1 A
ac1-2 A
ac1-3 A
ac1-4 B
ac1-5 C

ac2-1 C
ac2-2 B
ac2-3 E
ac2-4 D
ac2-5 A

ac3-1 G
ac3-2 D
ac3-3 D
ac3-4 A
ac3-5 B

initial state (time 1)
  ac1     B   ac2
    D     E     F
  ac3     H     I

time 2
  ac1   ac2     C
  ac3     E     F
    G     H     I

time 3
  ac1     B     C
  ac3   ac2     F
    G     H     I

time 4
  ac3   ac1     C
  ac2     E     F
    G     H     I

final state (time 5)
  ac2   ac3   ac1
    D     E     F
    G     H     I
```