HW 6

- two corrections / clarifications
  - #1 (Bellman-Ford) – starting vertex should be A
  - #3 (Prim) – starting vertex should be A
  - this has been corrected in the handout – use that if resubmitting
    - no penalty for a different choice on the initial handin

HW 6

- #1 – Bellman-Ford
  - go through every edge and update according to the current `dist` values for each iteration of the `repeat` loop
    - for simplicity go through the edges in alphabetical order – go through the vertices in alphabetical order and then each vertex's neighbors in alphabetical order
    - remember to consider both directions for undirected edges
  - document at least the current `dist` labels after the end of each iteration of the `repeat` loop, but showing more of your work doesn't hurt!

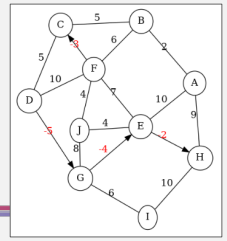
```
algorithm bellmanford(G,s):
  for all v in V do
    dist[v] ← ∞

  dist[s] ← 0

  repeat |V|-1 times
    updated ← false
    for all edges e=(v,u) in E do
      if dist[u] > dist[v]+w(v,u) then
        dist[u] = dist[v]+w(v,u)
        updated ← true
    if !updated then
      break
```

(the long reachabl that mar propaga

(can sto are no n

edge visit order –

AB	EA	HA
AE	EF	HI
AH	EH	IG
BA	EJ	IH
BC	FB	JE
BF	FC	JF
CB	FD	JG
CD	FE	
DC	FJ	
DF	GE	
DG	GI	
	GJ	



HW 6

- #5 – wedding guest seating arrangement
  - the desired goal is an acceptable seating arrangement – an assignment of people to tables avoiding pairs of guests on bad terms with each other – not (only) whether such an assignment is possible

HW 6

- graph modeling (#4-6)
  - the graph captures the problem setup and input (only)
    - make sure all of the information relevant to the solution of the problem is represented
      - e.g. for #4, winning the game means a series of moves to go from cell 1 to cell *n* – make sure all of the possible moves are represented in the graph
        - how are the dice rolls represented?
      - e.g. for #4, want to know the fewest dice rolls to win – make sure chutes and ladders aren't counted if these are separate moves from dice rolls
    - don't incorporate a solution strategy into the construction of the graph
      - e.g. for #4, omitting chutes from the possible moves because moving backwards won't help get to the end faster or only including cells at the ends of ladders because you'll always want to take a ladder
        - these may be correct strategies, but then you have to prove that…
    - don't build the graph to represent the solution or the objective
      - e.g. for #4, only have edges from cell *i* to cell *i*+1 plus for chutes and ladders because the player progresses through the cells in sequence (except due to chutes and ladders)

HW 6

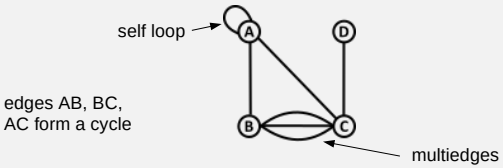
- graph modeling (#4-6)
  - "what is the solution to the problem in terms of the graph" means to translate the original problem's solution into the graph structures
    - e.g. for #6, you want a winning pickup order for the cards – how are the cards represented in the graph, and what does a winning pickup order correspond to?
    - not asking about the algorithm for solving the problem!
  - for "give an efficient algorithm for solving the problem" (only) identify the well-known problem or algorithm that can be used as a black box to solve the problem
    - you do not need to describe the algorithm for anything we've discussed in class: BFS, DFS, cut vertices, MST, Dijkstra's, …
    - if you need to modify the well-known algorithm or explain something about how to use it in this particular case, don't! (modify the graph instead)
    - make sure there's a clear connection between the well-known algorithm/problem and what the solution looks like in the graph
      - e.g. if you say Dijkstra's will solve the problem, make sure that you've identified that the solution to the problem in terms of the graph is a shortest path (and that you have a directed graph with weights > 0)

HW 6

- graph modeling (#4-6)
  - the graph properties (simple or not, sparse or dense, acyclic or not, etc) depend on the nature of the problem, not the solution or what is needed for a solution to exist
    - e.g. topological sort requires a DAG, but whether or not the graph you want to apply topological sort to is a DAG depends on the edges and whether it is possible for them to form cycles

HW 6

- *simple* means no self-loops or multiedges
  - a *self loop* is an edge that connects a vertex to itself
  - *multiedges* are multiple edges between the same pair of vertices
  - a simple graph can have cycles, or not
  - a simple graph can contain more than one edge (it most likely does!)
  - a simple graph can be directed, or not
    - in a directed graph, edges (u,v) and (v,u) are not multiedges



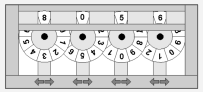
https://commons.wikimedia.org/wiki/File:Example_of_simple_undirected_graph_2.svg

HW 6

- embedded vs topological
  - *embedded* means the vertices and edges have geometric positions
    - a drawing of a graph is an embedding
  - for our purposes, the distinction is that *embedded* graphs have geometry and the geometry is significant to the task, while for *topological* graphs only the connections between vertices matter
    - e.g. a graph representing a road network is embedded if the real-world spatial layout matters – e.g. if a turn at an intersection is a right turn or a left turn – and topological if all that matters is which roads connect at intersections

HW 6

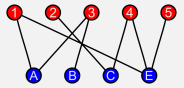
- implicit vs explicit
  - can you determine incident edges and adjacent vertices without directly storing what is connected to what?
    - yes → implicit, no → explicit
  - e.g. a graph representing a road network is *explicit* because the intersections and which roads connect which intersections must be stored
  - e.g. the graph used to find the fewest moves to get from one configuration to another is *implicit* because we can compute the adjacent configurations by adding/subtracting 1 from each value



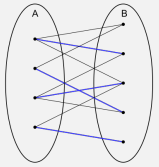
  - you can get in-between situations where some information must be stored but the rest can be computed
    - e.g. the forbidden configurations

HW 6

- be careful not to mix up similar-sounding terms and tasks
  - a *bipartite graph* is one whose vertices can be divided into two groups so that no edge connects two vertices in the same group
  - a *2-coloring* is an assignment of two colors to vertices so that no edge connects two vertices of the same color
    - a graph with a 2-coloring is bipartite, and a bipartite graph has a 2-coloring – finding a 2-coloring (or failing to do so) is a way of showing that a graph is bipartite (or not)
  - *bipartite matching* refers to finding a set of edges in a bipartite graph such that no edges in the set share a vertex
    - typically the goal is the largest such set



bipartite graph with a 2-coloring



blue edges form a maximum matching

https://commons.wikimedia.org/wiki/File:Simple_bipartite_graph;_two_layers.svg
https://commons.wikimedia.org/wiki/File:Halls_theorem_matching_graph_theory.svg