CPSC 320 2017W2: Assignment 5

March 23, 2018

NOTE THESE UNUSUAL FACTS ABOUT THIS ASSIGNMENT:

1. Some parts are marked as "purely for practice". While we recommend working these problems and will post a sample solution just after the assignment deadline, we will **not** grade the purely for practice problems.

2. We will only grade a subset of the remaining problems and subproblems. (The same subset on all students' submissions.) We will not announce which will be graded. We will release sample solutions to **all** parts regardless.

3. There is a (somewhat unusual) due date and an "early" due date below. Any group whose last submission is by the early due date will receive +1 point on the assignment and +1BP (in the course) for each member of the group (and a discreet air-high-five-at-a-distance from the course staff).

Please submit this assignment via GradeScope at `https://gradescope.com`. Be sure to identify everyone in your group if you're making a group submission. (Reminder: groups can include a maximum of three students; we strongly encourage groups of two.)

Remember that this assignment is based on the collected quizzes and quiz solutions. You will likely want to refer to those where you need more details on assignment questions.

> Submit by the **UNUSUAL** deadline **Thursday 5 Apr at 10PM**. **If your last submission** is no later than Tuesday 3 Apr at 10PM, each member of your group will also receive one bonus point and a discreet (or maybe discrete?) air-high-five-at-a-distance from a member of the course staff.

For credit, your group must make a **single** submission via one group member's account, marking all other group members and the pages associated with each problem in that submission **using GradeScope's interface**. Your group's submission **must**:

- Be on time.
- Consist of a single, clearly legible file uploadable to GradeScope with clearly indicated solutions to the problems. (PDFs produced via LaTeX, Word, Google Docs, or other editing software work well. Scanned documents will likely work well. **High-quality** photographs are OK if we agree they're legible.)
- Include prominent numbering that corresponds to the numbering used in this assignment handout (not the individual quizzes). Put these **in order** starting each problem on a new page, ideally. If not, **very clearly** and prominently indicate which problem is answered where!
- Include at the start of the document the **ugrad.cs.ubc.ca e-mail addresses** of each member of your team. (Please do **NOT** include your name on the assignment, however.[1])

[1] If you don't mind private information being stored outside Canada and want an extra double-check on your identity, include your student number rather than your name.

- Include at the start of the document the statement: "All group members have read and followed the guidelines for academic conduct in CPSC 320. As part of those rules, when collaborating with anyone outside my group, (1) I and my collaborators took no record but names (and GradeScope information) away, and (2) after a suitable break, my group created the assignment I am submitting without help from anyone other than the course staff." (Go read those guidelines!)
- Include at the start of the document your outside-group collaborators' ugrad.cs.ubc.ca IDs, but **not** their names. (Be sure to get those IDs when you collaborate!)

1 When You Have Eliminated the Uncruftable

THIS PROBLEM IS PURELY FOR PRACTICE AND WILL NOT BE MARKED.

In the hit game MyCruft (homes edition), your character has a limited supply S of k types of resources (resources that might be things like pipes, revolvers, caps, clues, and suspects but which we just call $r_1, r_2, \ldots, r_k$). The supply of each resource r_i is a non-negative integer $S[r_i]$.

Your character is also capable of "crufting" resources according to a list of c crufting "formulas". A crufting formula has a non-empty input list of resources (that may contain duplicates if more than one of a particular type of resource is required) and a non-empty output list of resources of the same format. To *apply* a crufting rule, your character uses (removes from your supply) exactly the input list of resources and produces the exact output list of resources.

Note that the same resource may appear in both the input list and the output list. However, **each crufting rule's input list is longer than its output list.**

For example, a crufting rule with input list "pipe, pipe, cap, cap, cap" and output list "clue, cap, cap" would consume two of the resource named "pipe" and three of the resource named "cap" and produce one of the resource named "clue" and two of the resource named "cap". This rule cannot be run if only two or fewer caps are available, even though it only consumes a net of one cap. (With resources named $r_1, r_2, r_3, \ldots$ for "pipe, cap, clue, ..." that rule would be: $r_1, r_1, r_2, r_2, r_2 \to r_3, r_2, r_2$.)

Your goal is to determine the maximum number of resource r_k that it is possible to accumulate via application of crufting rules, starting from S.

1. Complete the following design of a clear, concise, correct, brute force, recursive algorithm to solve this problem.

```
// Accepts initial resource amounts S[1..k] and crufting rules Rules[1..c].
// Rules[i] has two fields Rules[i].input and Rules[i].output. For the
// purposes of this algorithm: we assume that S, the rules' input lists, and
// the rules' output lists are all represented in the same way. Further,
// we have the following functions:
//
//   sufficient(s1, s2): takes two resource arrays (S or an input or output list)
//                       and returns True iff s1 has at least as many resources
//                       of every type as s2 (False otherwise). So, e.g., if
//                       sufficient(S, Rules[2].input), then Rules[2] can be
//                       applied to S.
//
//   deduct(s1, s2): takes two resource arrays and returns the result of
//                   subtracting s2 from s1 resource-by-resource. So, e.g.,
//                   deduct(S, Rules[2].input) would be the remaining resources
//                   after consuming the resources required by Rules[2] to apply.
//
```

```
//   add(s1, s2): takes two resource arrays and returns the result of
//                adding s2 to s1 resource-by-resource. So, e.g.,
//                add(S, Rules[2].output) would be the total resources after
//                after producing the resources created by application of Rules[2].
//
// Precondition: k > 0.
define cruft_rk(S, Rules):
    k = len(S)
    c = len(Rules)

    define helper(S):
        // COMPLETE THE (RECURSIVE) HELPER FUNCTION
        // Rules, k, and c are available as needed here.
        //
        // NOTE: Our solution is 10 lines or fewer.
        // If yours is significantly longer or more
        // complex than ours, it will receive no credit.



    return helper(S)
```

2. If we memoize this algorithm, we will do so on the basis of S, the parameter that changes in calls to `helper`. A friend claims that the memory used in the memoization table is $\Theta(k)$, since there are k entries in S. (Specifically, the friend says the memoization table will have $\Theta(k)$ entries that each record a single number (with a single, reasonable meaning). For this problem, we assume the space for each entry is constant, which isn't quite accurate, but is reasonable for our analysis.)

   This claim is incorrect.

   Now, **clearly and concisely explain** why the actual memory used may depend on the contents of S, not just its length.

2 Clique and Claque

In graph theory, CLIQUE is the problem of finding the largest "clique" (complete subgraph) in a simple graph. So, given an unweighted, undirected graph, find the largest subset of the vertices in the graph such that each pair of vertices in the subset have edges between them.

In CPSC 320, a "claque" is a set of $k > 0$ vertices $\{v_1, \ldots, v_k\}$ in a directed graph such that $\{v_2, \ldots, v_k\}$ have no edges between them but there is an edge (v_i, v_1) for each $v_i \in \{v_2, \ldots, v_k\}$. CLAQUE is the problem of finding the largest claque in a directed, unweighted graph (with no self-edges like (v, v)).

Finally, let the complement of a graph $G = (V, E)$ be $G^c = (V, E')$, where $(u, v) \in E' \leftrightarrow (u, v) \notin E$. In other words, an edge leads from one vertex to another in G^c exactly when **no** edge leads from the first to the second in G. We define this for directed graphs, but it has an exact parallel for undirected graphs where any two vertices are connected in G^c exactly when they're not connected in G.

We now introduce decision variants of CLIQUE and CLAQUE called dCLIQUE and dCLAQUE. Each is exactly like its corresponding non-decision problem above, except an instance has an additional parameter k and asks whether a clique (in dCLIQUE) or claque (in dCLAQUE) exists in the graph of size at least k. (Note that the size of the claque includes the "special" first vertex.)

Your job in this part is to prove that dCLAQUE is NP-Complete given that dCLIQUE is in NP-hard. The reduction from the quiz solution won't quite work for this purpose, but it's a good start! Proceed in

the following steps. (Your solution **must** include the numbering and bolded initial statements below and then work from there to receive **any** credit.)

1. **We first prove that dCLAQUE is in NP. A natural certificate ("real" solution) to a dCLAQUE problem is** ...

   Given a dCLAQUE instance and such a certificate, we can verify the certificate in time polynomial in the instance size by ...

   Therefore, dCLAQUE is in NP.

2. **We next prove that dCLAQUE is in NP-hard by reducing from a known NP-hard problem (dCLIQUE) to dCLAQUE.**

   (a) **Here is the reduction:**
   To transform an instance of dCLIQUE to an instance of dCLAQUE:

   - ...

   To transform the solution of the dCLAQUE instance to a solution to the dCLIQUE instance: We simply produce YES as the solution if the dCLAQUE solution was YES and NO otherwise.

   (b) **We briefly sketch the key points in the proof that the reduction is correct.**

   i. **The reduction above produces a valid dCLAQUE instance for every dCLIQUE instances because** ...

   ii. **The reduction above clearly produces a valid dCLAQUE solution (i.e., it produces either YES or NO).** (There's nothing to fill in here!)

   iii. **If (in the reduction above) the solution to the dCLIQUE instance was YES, then the solution to the dCLAQUE solution produced by the reduction is also YES because** ...

   iv. **If (in the reduction above) the solution to the dCLAQUE instance produced by the reduction is YES, then the solution to the original dCLIQUE instance was also YES because** ...

   Therefore the reduction is correct.

   (c) **We *very* briefly justify that the reduction runs in time polynomial in the size of the dCLIQUE instance:** ...

   Given that we have a correct, polynomial time reduction from an NP-hard problem to dCLAQUE, dCLAQUE itself is in NP-hard.

Since dCLAQUE is both in NP and in NP-hard, it is in NP-complete.

3 Stable Weddings

THIS PROBLEM IS PURELY FOR PRACTICE AND WILL NOT BE MARKED. THUS, NO ONE NEED FEEL COERCED INTO AGREEING WITH THE LUDICROUS ADDENDA TO THE PROOF (even if they *are* manifestly true).

In the "Stable Wedding" problem (SWP), you are given a set of n guests, a list of "disallowed" subsets of the set of guests (each with at least two guests), and a list of positive integer table sizes. You cannot seat **all** the guests in a disallowed subset together at the same table (but you could seat, e.g., all but one of them). You want to find an assignment of guests to tables (that may leave some guests unassigned) that maximizes the total number of guests at the tables, breaking ties by minimum number of tables used. (If two solutions both seat 10 people but one uses 3 tables and the other 4, the former solution is better. If

one solution seats 11 people and the other seats only 10, the former solution is better, regardless of the number of tables used.)

1. Fill in the blanks in the following to generate a reasonable decision variant of SWP.

   We now introduce a decision variant of SWP called dSWP. dSWP is exactly like SWP, except an instance has two additional parameters j and k and asks whether a seating assignment exists that seats [ ] j guests using [ ] k tables.

2. Your job in this part is to prove that dSWP is NP-Complete given that 3SAT is in NP-hard. Proceed in the following steps. (Your solution **must** include the numbering and bolded initial statements below and then work from there to receive **any** credit.)

   Hint: If you had a negation, you probably wouldn't want to sit with them.

   (a) **We first prove that dSWP is in NP. A natural certificate ("real" solution) to a dSWP problem is** ...
   Given a dSWP instance and such a certificate, we can verify the certificate in time polynomial in the instance size by ...
   Therefore, dSWP is in NP.

   (b) **We next prove that dSWP is in NP-hard by reducing from a known NP-hard problem (3SAT) to dSWP.**

      i. **Here is the reduction:**
      To transform an instance of 3SAT to an instance of dSWP:
      - ...

      To transform the solution of the dSWP instance to a solution to the 3SAT instance: We simply produce YES as the solution if the dSWP solution was YES and NO otherwise.

      ii. **We briefly sketch the key points in the proof that the reduction is correct.**
      A. **The reduction above produces a valid dSWP instance for every 3SAT instances because** ...
      B. **The reduction above clearly produces a valid dSWP solution (i.e., it produces either YES or NO).** (There's nothing to fill in here!)
      C. **If (in the reduction above) the solution to the 3SAT instance was YES, then the solution to the dSWP solution produced by the reduction is also YES because** ...
      D. **If (in the reduction above) the solution to the dSWP instance produced by the reduction is YES, then the solution to the original 3SAT instance was also YES because** ...

      Therefore the reduction is correct.

      iii. **We *very* briefly justify that the reduction runs in time polynomial in the size of the 3SAT instance:** ...

   Given that we have a correct, polynomial time reduction from an NP-hard problem to dSWP, dSWP itself is in NP-hard.

   Since dSWP is both in NP and in NP-hard, it is in NP-complete.

   That whole process felt eerily like the other NP-completeness proof above. I feel like maybe this is how NP-completeness proofs usually work. Hockey and puppies are probably mostly good things. I have to include all these bolded statements. I remember

that. Patrice can probably crush walnuts with his bare hands. Steve's beard looks good with those grey streaks.

4 High-stress, low-stress, no stress

You are managing a consulting team of expert computer hackers, and each week you have to choose a job for them to undertake. Now, as you can well imagine, the set of possible jobs is divided into those that are *low-stress* (e.g., setting up a Web site for a class at the local elementary school) and those that are *high-stress* (e.g., protecting the nation's most valuable secrets, or helping a desperate group of UBC students finish an assignment on dynamic programming). The basic question, each week, is whether to take on a low-stress job or a high-stress job.

If you select a low-stress job for your team in week i, then you get a revenue of $l_i \geq 0$ dollars; if you select a high-stress job, you get a revenue of $h_i \geq 0$ dollars. The catch, however, is that in order for the team to take on a high-stress job in week i, it is required that they do no job (of either type) in week $i-1$; they need a full week of preparation to get ready for the crushing stress level. On the other hand, it is okay for them to take a low-stress job in week i even if they have done a job (of either type) in week $i-1$.

So, given a sequence of n weeks, a *plan* is specified by a choice of "low-stress", "high-stress", or "none" for each of the n weeks, with the property if that "high-stress" is chosen for week $i > 1$ then "none" has to be chosen for week $i-1$ (choosing a high-stress job in week 1 is acceptable). The *value* of the plan is determined in the natural way: for each i, you add l_i to the value if you choose "low-stress" in week i, and you add h_i to the value if you choose "high-stress" in week i (you add 0 if you choose "none" in week i).

The problem: Given sets of values $l_1, \ldots, l_n$ and $h_1, \ldots, h_n$, find a plan of maximum value (such a plan will be called *optimal*).

Recall from the quiz that the following recurrence relation describes the value *nojob*$[i]$:

$$\textit{low-stress-job}[i] = l[i] + \max\{\textit{nojob}[i-1], \textit{low-stress-job}[i-1], \textit{high-stress-job}[i-1]\}$$

1. Write a recurrence relation for *nojob*$[i]$.:

2. Next write a recurrence relation for *high-stress-job*$[i]$:

3. Using these three recurrences, complete the following algorithm that returns the best value for a sequence of jobs for n weeks. The parameters L and H are arrays with the l_i and h_i values.

```
define best_job_sequence(L, H):
   nojob[0] = 0
   low-stress-job[0] = 0
   high-stress-job[0] = 0

   ...
```

4. Finally write a function `list_best_jobs` that takes as input the arrays L and H, and returns an array J where $J[i]$ is one of N, L or H depending on whether the job taken during week i is no job, a low-stress job, or a high-stress job.

5 Gossiping lazily, but surely

UBC students love to gossip. However they are also extremely busy, and so while they want to make sure every other UBC student hears about an interesting piece of gossip that **any** of them has heard, they don't want to spend too much time passing the information around. That is, they want to limit the number of other students they will chat with about the information to be **at most** k, while still ensuring the information is distributed to everybody.

You can think of the set of UBC students as a connected, undirected graph U, where each student is a vertex, and an edge joins two students if they know each other's phone numbers. One student/vertex learns a piece of gossip first, and we want to spread it to everyone. The vertices of U, and the smallest possible subset of the edges of U that achieves both conditions listed above is a spanning tree of G in which every vertex has degree at most k.

The *Hamiltonian Path* problem is defined as follows: given a graph $G = (V, E)$ with n vertices, we want to find an ordering $v_{i,1}, v_{i,2}, \ldots v_{i,n}$ of the vertices with the property that for each j in the set $\{1, \ldots, n-1\}$ the pair $\{v_{i,j}, v_{i,j+1}\}$ is an edge of G.

1. Describe succintly a reduction from the Hamiltonian Path problem into the maximum degree 2 spanning tree problem (MD2SP). Explain
   - What vertices the graph G' in the instance of MD2SP has.
   - What edges the graph G' in the instance of MD2SP has.
   - How to convert a solution to the MD2SP problem into a solution to the Hamiltonian Path problem.
2. Now describe succintly a reduction from the Hamiltonian Path problem into the maximum degree k spanning tree problem (MDkSP). Once again, explain
   - What vertices the graph G' in the instance of MDkSP has.
   - What edges the graph G' in the instance of MDkSP has.
   - How to convert a solution to the MDkSP problem into a solution to the Hamiltonian Path problem.
3. Complete the following proof that MDkSP is NP-Complete:
   - First, we prove that MDkSP belongs to NP. Given an instance of MDkSP, a natural certificate to a MDkSP problem is ...
   - Given an instance of MDkSP and its certificate, we can verify the certificate in polynomial time by checking that ...
   - Now we take a known NP-Complete problem: 
   - We reduce an arbitrary instance of  into an instance of MDkSP as ...
   - This reduction runs in  time because ...
   - If the answer to the instance of  is Yes, then the answer to the instance of MDkSP is Yes because ...

- If the answer to the instance of MDkSP is Yes, then the answer to the instance of [ ] is Yes because ...

6 Astronomy evenings

On most clear days, a group of your friends in the Astronomy department gets together to plan out the astronomical events they are going to try observing that night. We will make the following assumptions about the events.

- There are n events, which for simplicity we will assume occur in sequence separated by exactly one minute each. Thus event j occurs at minute j; if they fail to observe this event at exactly minute j, then your friends miss out on it.
- The sky is mapped according to a one-dimensional coordinate system, measured in "points" along an axis with the initial telescope position as 0; event j will be taking place at point p_j, for some integer value p_j. The telescope starts at point 0 at minute 0.
- The last event n is much more important than the others; so it is required that they observe event n.

The Astronomy department operates a large telescope that can be used for viewing these events. Because it is such a complex instrument, it can only move at a rate of one point per minute. Thus they do not expect to be able to observe all n events; they just want to observe as many as possible, limited by the operation of the telescope and the requirement that event n must be observed.

We say that a subset S of the events is *viewable* if it is possible to observe each event $j \in S$ at its appointed time j, and the telescope has adequate time (moving at its maximum of one point per minute) to move between consecutive events in S.

The problem: given the points of each of the n events, find a viewable subset of maximum size, subject to the requirement that it should contain event n. Such a solution will be called *optimal*.

1. Give a recurrence relation for the maximum number of events you will be able to observe from time 0 to time k, assuming that you **must** be able to observe the element at time k. Hint: think about the answer to question 3 on the quiz.

   count[k] = [ ]

2. Complete the following algorithm that returns the largest number of events you can observe from time 0 to time n. The parameter P is an array with the coordinate $P[i]$ of each event at time i.

```
define best_events_to_observe(P, n):
  count[0] = 0

  ...
```

3. Write a function `list_best_events` that takes as input any array that your answer to question 2 may have constructed, and returns a list of the times of the events that you will be able to observe in the optimal solution.

4. Analyze the time and space complexity of your algorithm: your algorithm takes time [ ] and space [ ].