

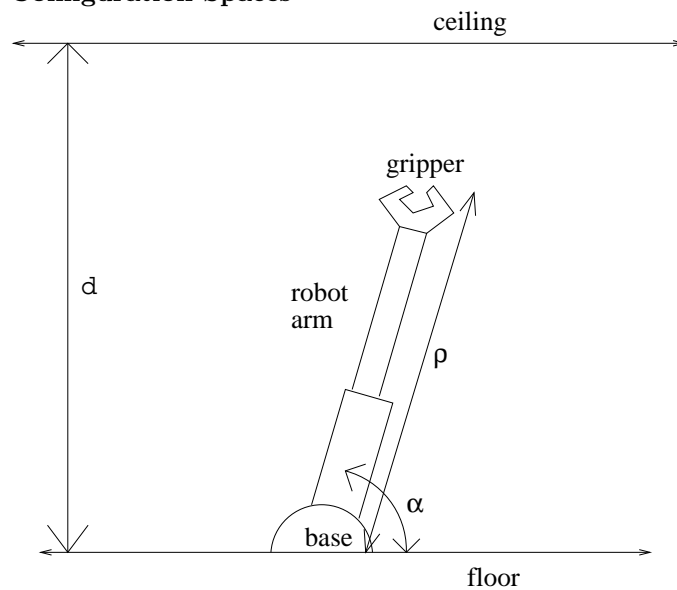
CS 221 Problem Set #1 Solutions: Search, Motion Planning, CSPs

Due by 9:30am on Tuesday, October 13. Please see the course information page on the class website for late homework submission instructions. SCPD students can also fax their solutions to (650) 725-1449. We will not accept solutions by email or courier.

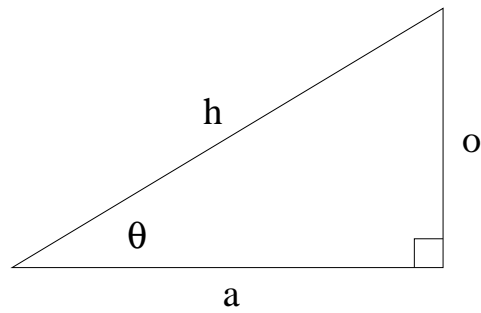
Written part (70 points)

NOTE: These questions require thought, but do not require long answers. Please try to be as concise as possible.

1. [10 points] Configuration Spaces



Consider the robot arm pictured above with two degrees of freedom, operating in a two dimensional workspace. The robot arm has a revolute joint and a prismatic joint. The revolute joint has a range of $0 \leq \alpha \leq \pi$, where α is the angle of the arm relative to the floor. The prismatic joint has a range of $\rho_{min} \leq \rho \leq \rho_{max}$, where ρ is the length of the arm from the base to the gripper. The ceiling is a distance d from the floor, with $\rho_{min} < d < \rho_{max}$. The width of the arm and gripper may be considered negligible.

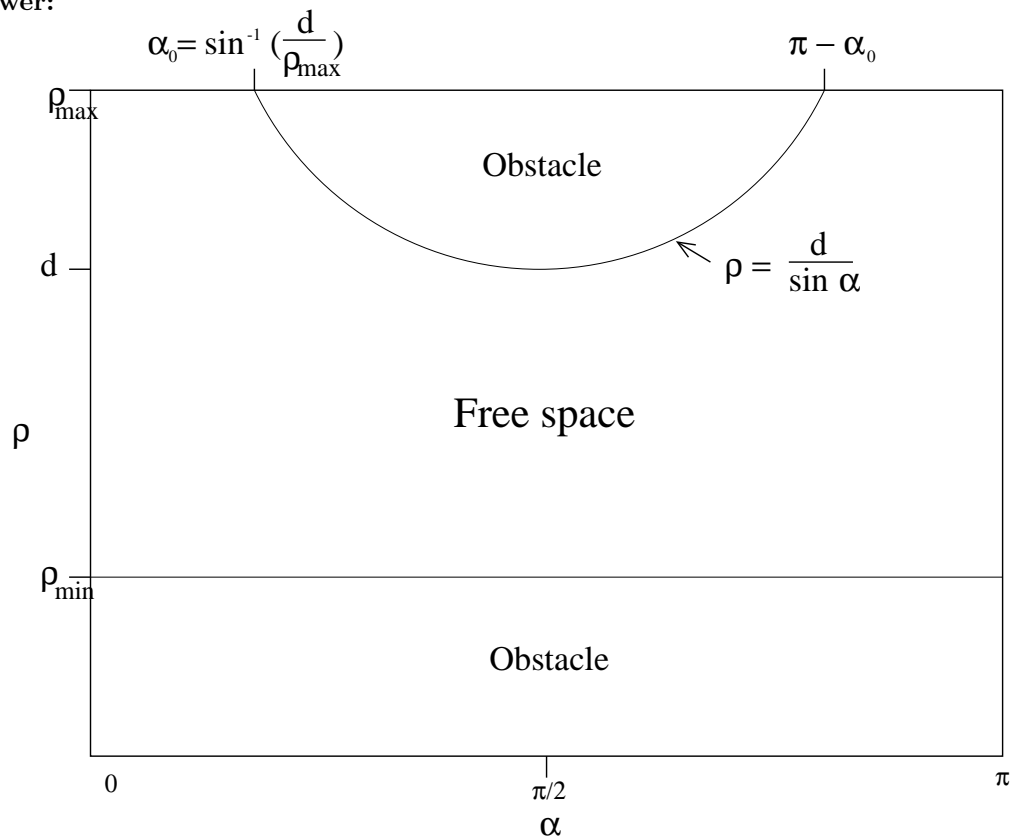


$$\sin \theta = o/h \quad \cos \theta = a/h$$

Draw the configuration space of the robot arm, using α and ρ as the coordinates of the configuration space (i.e., the horizontal and vertical axes in your figure should be labeled α and ρ).

Please specify the coordinates of the important points of the obstacles in configuration space (e.g. leftmost point, rightmost point, etc.). Also, while a freehand drawing is acceptable, please make sure that the shape of the obstacle is clear from your drawing.

Answer:



Common Mistakes:

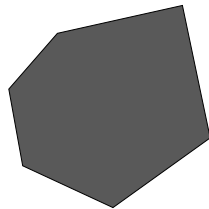
In general everyone understood this question, at least to the extent that they could draw the obstacles, if not also calculate the critical points. A few people omitted the "lower" obstacle (which arises due to the fact that the arm's minimum length is ρ_{\min}). A number of people lost a point by providing the equation of the boundary only (we ask for the coordinates of the points).

Error Codes:

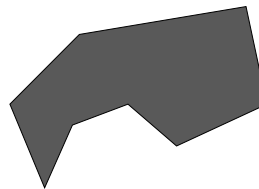
- E1 (-1) minimum arm-length obstacle (below ρ_{\min}) omitted or incorrectly drawn
- E2 (-2) drawing in polar coordinates (this is not α, ρ coordinates)
- E3 (-2) any characterization of the important points in the drawing missing
- E4 (-2) semi-circle obstacle boundary extending to π and 0 (not counting missing interesting points then)
- E5 (-1) ceiling boundary obstacle specified by an equation but not the coordinates of the intersection points.
- E6 (-1) ceiling boundary obstacle incorrectly drawn (we accepted every shape that vaguely looked like a circle but not straight lines with a sharp angle at $\alpha = 0$)
- E7 (-1) small mathematical errors when describing the points, and the boundary if it was specified.
- E8 (-6) completely wrong answer (i.e. drawing the task space)
- E9 (-1) missing boundary, e.g., floor

2. [14 points] **Optimization Search / Configuration Spaces**

We consider the problem of moving a robot in various two dimensional (2D) configuration spaces with obstacles. We start with a simple configuration space, with only convex obstacles, and then make it more complicated by allowing non-convex obstacles as well.



convex obstacle



nonconvex obstacle

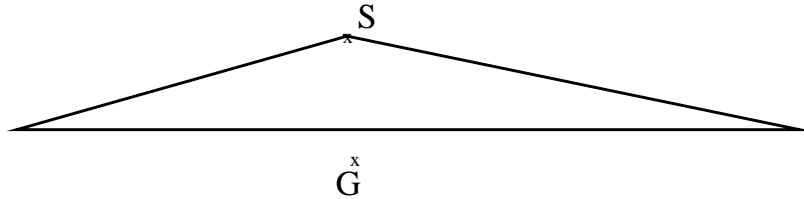
A discrete search space for this planning problem can be defined using the *visibility graph method*. In this method, we place landmarks in configuration space at the initial position of the robot, the goal position of the robot, and the *vertices* of the polygonal obstacles. Further, the search operators only allow the robot to walk in a straight line between two of these points: a state s in the search space is connected to any other state s' which can be reached from s by walking along a straight line either completely in free space or along the boundary of an obstacle (i.e., s is connected to s' if s' is "visible" from s).

Assume that our robot navigates this space using a simple greedy hill-climbing search with a straight-line distance heuristic function. (Note that the robot climbs "down" in that the heuristic function decreases as the robot approaches the goal. Nevertheless, we'll continue to use the "climbing" metaphor, as we did in class.)

We assume that the robot is allowed to move in any direction, but is not allowed rotation.

- (a) [7 points] For this part, assume that the robot is point-sized, so that the configuration space is a 2D space which is identical to the workspace. A local maximum is a state that is closer to the goal than any of its successors. If the workspace contains only convex polygonal obstacles, is it possible for the robot to get stuck on a local maximum? Either explain why not, or present an example.

Answer: Even though all of the obstacles in the plane are convex, the robot can still get stuck using hill climbing, as in the following example:



In this case, the initial position is much closer to the goal (in terms of straight-line distance) than either of the two endpoints of the obstacle, so the robot will simply stay there.

Grading criteria:

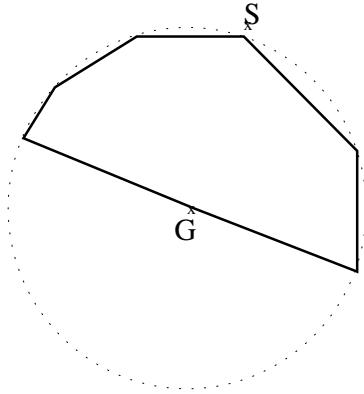
- a0 0 points for saying no.
 - a1 2 points for saying yes but providing an incorrect example with no/incomplete explanation.
 - a2 4 points for a correct example with a slightly incorrect/poor explanation.
 - a3 5 points for an example where the robot would oscillate between two points (a local *maximum* is really a single point).
 - Full points for a correct example.
- (b) [7 points] Again assume that the robot is point-sized. If we allow arbitrary polygonal obstacles, can there be any plateaus in this space? Either explain why not, or present an example.

For this problem, define a plateau to be a contiguous region of 4 points or more where the points all have the same heuristic value (so that it is locally impossible to determine the direction to the goal). There can be other neighboring points where you “fall off the edge” of the plateau, so that the heuristic value is strictly worse. But the plateau itself has to consist of a contiguous set of points with the same value.

Answer: A plateau consists of at least one point n such that

- i. n and some of its neighbors have the same heuristic value, (and all other neighbors have a worse heuristic value) so that the algorithm cannot tell which is the right direction to proceed.

This search space contains plateaus both for convex and nonconvex obstacles. Consider a polygon some of whose vertices are arranged on a the perimeter of a circle, with the goal being at the center of the circle:



Then all the points in the perimeter are equally far from the goal. Note that this construction relies on the definition of our search space, since only the vertices are equidistant from the goal; the points on the edges connecting them are not.

Grading criteria:

b0 0 points for saying no.

b1 2 points for saying yes but providing an incorrect example.

b2 4 points for an example with four equidistant points that do not form a plateau (a common mistake was to draw four equidistant points from which the goal is visible, in which case there is a clear downhill direction)

- Full points for a plateau with at least 4 contiguous points that are equidistant from the goal.

3. [18 points] **Search Space Formulation** Consider the problem of a student planning an entire course of studies at Stanford in advance. The desired outcome is an assignment of classes to quarters (1st quarter, 2nd quarter, etc.) which satisfies the requirements of the major. To simplify the problem, assume the following:

- There is some set of classes C such that the student must take each of the classes in C exactly once (i.e., there are no alternatives of the form: you must take either CS106A and CS106B or CS106X).
- For each class c_i , there is a (possibly empty) set $P_i \subseteq C$ of prerequisite classes that the student must take strictly before s/he takes c_i .
- The student must take at least one and at most four classes each quarter.

(a) [8 points] Provide a detailed representation of your search space. Your description must be precise, i.e. you should provide an exact specification of the components of the state description (state space, initial state, operators, goal test), of the constraints under which each operator can be applied, and of the effects of each operator on the state components. You may use either English or pseudocode.

Make sure that your formulation of the search space has no problem with repeated states, i.e., that no node can be reached from the initial state by two different paths.

Answer: We can formulate the class scheduling problem as a search problem by letting states represent partial schedules (schedules that do not necessarily include every class in C), and letting operators add classes to these schedules. Given this general idea, there are a number of reasonable ways to define the state space. We will discuss a formulation in which a state is a list of pairs (c, q) , where c is a class and q is a natural number representing the quarter in which the student will take that class. We will define our

operators such that the quarters in each state are consecutive and each quarter has 1 to 4 classes assigned to it. One could also define a state as a list of pairs (C_q, q) , where C_q is the set of classes to be taken in quarter q .

The initial state is an empty set; that is, a schedule in which no classes are assigned to any quarters.

We have two kinds of operators: one adds a class to the last quarter in the current schedule, and one adds a class to a new quarter that is appended to the end of the schedule. We are not allowed to add a class that is already in the schedule, or one whose prerequisites are not already scheduled for previous quarters. Also, we cannot have more than 4 classes scheduled for any one quarter.

We must also deal with the problem of repeated states. Now, because our operators only add classes to a schedule, we will never see the same state twice on a search path. However, we can still get repeated states; for example, we can get the schedule $\{\text{English}, 1\text{st quarter}\}$, $\{\text{Compilers}, 1\text{st quarter}\}$ either by adding English first or by adding Compilers first.

There are a number of ways to deal with this problem. An easy way is to restrict the operators so that classes in the same quarter are scheduled in alphabetical order. That is, we cannot schedule a class in quarter q if a class that's alphabetically later has already been scheduled in quarter q . We could also avoid this problem by using the alternative state space definition mentioned above, and just having one kind of operator that schedules an entire quarter's worth of classes at once. This would result, in general, in a shallower search tree, but a larger branching factor.

Here is a formal definition of the operators, using alphabetical order to avoid repeated states. Let q^* be the highest number quarter in the schedule corresponding to a state. The legal successors of this state are those that add a pair (c, q') such that:

- c is not in the schedule,
- all of c 's prerequisites are paired with quarters less than q' in the schedule,
- q' is either q^* or $q^* + 1$,
- if $q' = q^*$ then the total number of pairs $(*, q')$ in the new schedule is at most 4, and
- no alphabetically later class is already scheduled in quarter q^* .

The goal test checks that the list contains all the required classes. The definition of the state space and operators precludes a schedule which contains all the classes but does not satisfy the prerequisite constraints.

As an alternative search space formulation, each state will be a list of tuples $\langle \text{quarter}, \text{classSet} \rangle$ corresponding to consecutive quarters. The list represents a partial schedule. For example, one state in the search space might be

$$\langle \text{Quarter}_1, \{cs109, cs140, cs157\} \rangle, \langle \text{Quarter}_2, \{cs248, cs145\} \rangle$$

The constraints for each tuple are:

- The size of classSet is at least 1 and at most 4.
- Each class in classSet is drawn from the initial list of required classes, and only appears once.
- For each class c in classSet , all of c 's prerequisites must be in the classSet of a tuple with an earlier quarter .

The constraints for the entire state are the constraints for each tuple, plus:

- The *quarter* of the i th tuple in the list must have $quarter = Quarter_i$. (Otherwise, the search space will have repeated states when the same set of tuples is added to the list in different orders.)
- Each class does not appear in more than one *classSet*.

The operator adds a new tuple to a state's list, subject to the above constraints. The initial state is an empty list of tuples. The goal test checks that all classes appear in the list, or, more formally, that the union of *classSets* in all tuples of the list is equal to the set of all required classes. Note that the goal test does not need to check any of the constraints listed above, because the initial state satisfies all constraints, and we have defined the operator so that no successor state can violate them.

Grading criteria:

3a1 (-1 point each) One of the following is defined with insufficient precision: state space, initial state, operators, goal test, constraints.

3a2 (-3 points) The operators may produce repeated constraints.

3a3 (-1 point) The search space is incomplete. (A common way for this to happen is if each quarter is only allowed to have precisely 4 classes)

3a4 (-2 points each) An item in the search space formulation is missing entirely.

3a5 (-2 points) Solution depends on the search algorithm maintaining extra data structures, such as a list of all nodes ever visited.

3a6 (-3 points) Search space formulation only tests whether a complete schedule is possible, does not return a complete specification of which classes to take each quarter

3a7 (-1 point) Formulation will result in inefficient behavior. i.e., prerequisites are only checked in the goal test.

- (b) [**3 points**] Somewhat shortsightedly, the student's only aim is to complete the degree in the shortest possible time, thereby hopefully minimizing the total amount of money spent on tuition. Tuition is charged per quarter, not per unit, and all quarters cost the same amount. Specify a cost function for the operators in your space so that the cost of a solution will correspond exactly to the student's preferences. **Answer:** We assumed that tuition is charged per quarter, not per unit. Since the cost function must reflect the student's real-world cost, the cost of a schedule should correspond to the number of quarters in it. Thus, the operators above can have one of two costs. If the pair added is $(c, q^* + 1)$, then we have added a new quarter, and the cost of the operator is 1. Otherwise, we have scheduled a class in an already existing quarter, and the cost of the operator is 0.

For the alternative formulation, the operator always adds one quarter to the schedule. So the cost of the operator is 1.

Grading criteria:

3b1 (-3 points) Solution does not correspond exactly to student's preferences.

- (c) [**7 points**] Construct a reasonable and admissible heuristic for this search space, given your costs in 3b. (For example, a zero heuristic function is admissible but not reasonable; for different reasons, a heuristic function that computes the optimal schedule is not reasonable either.) Provide an exact description of your heuristic function, and explain why it's admissible, why it's a reasonable heuristic, and how you would compute it efficiently given a state in your search space. (Hint 1: As discussed in lecture, try relaxing some of the constraints on your operators. Hint 2: There are multiple, equally good answers to this problem.)

Answer: The heuristic function is an estimate of the remaining cost until the goal (graduation) is reached. In class, we discussed one approach for designing admissible heuristic functions: removing constraints from the operators, and estimating the cost to the goal using those relaxed operators. Here, there are two main constraints on the operators:

- (a) you can't schedule more than 4 classes a quarter;
- (b) you can't schedule a class before its prerequisites.

We can eliminate (a) or (b). (We can also eliminate both, but there's no need.) It turns out that both of these options lead to very intuitive admissible heuristics.

If we eliminate (b), our operators allow us to schedule any class in an empty slot, regardless of the prerequisites. Therefore, the cost is essentially the number of remaining classes, divided by 4 (since we can still schedule only 4 classes per quarter). More formally, the heuristic function h_a is the number of unscheduled classes *minus the number of empty slots in the current quarter* q^* , divided by 4 (and rounded up). This heuristic is very easy to compute: we know the length of the current list of scheduled classes, so we can subtract this from $|C|$ to get the number of unscheduled classes. Figuring out the number of empty slots in the current quarter is easy, since classes scheduled for the current quarter are at the end of the list; then we just have to do a division.

If we eliminate (a), our operators allow us to schedule as many classes a quarter as we want, so long as we satisfy the prerequisite requirements. Therefore, the cost is the longest chain of prerequisites among the unscheduled classes. More formally (and accounting for the quarter currently being scheduled), the heuristic function h_b is the length of the longest chain of prerequisites among the set of unscheduled classes and the classes scheduled for q^* , minus 1. Computing this heuristic is a bit more difficult: we need to iterate over the unscheduled classes and the classes scheduled for q^* , and find the length of the chain of prerequisites leading to each one. This can be done with a recursive algorithm, augmented with a lookup table so we don't have to compute the length of the prerequisite chain for a given class more than once.

Since both h_a and h_b are admissible, then so is their maximum $\max(h_a, h_b)$. This is probably a better heuristic than either of the two. We gave 1 point extra credit to people who used this maximum.

Both of the above heuristics will also work for the alternative formulation, with minor changes. For h_a , we ignore the number of empty slots in the current quarter, since the operator never adds more classes to current quarter. For h_b , we only look at the length of the longest chain of prerequisites among the set of unscheduled classes.

Grading criteria for part (c):

- 3c1** (-1 point) Does not subtract number of empty slots in current quarter (for the one-class-at-a-time formulation only)
- 3c2** (-2 points) Heuristic is not as tight as it could be given the relaxation used. i.e., did not use rounding, used floor, or subtracted some constant to avoid making the bound tight.
- 3c3** (-7 points) The solution is completely incorrect.
- 3c4** (-2 points) Heuristic is sensible but does not correspond to your operators (i.e., your heuristic refers to the number of unassigned classes, but your operators assume all classes are assigned and then swaps their locations)
- 3c5** (-4 points) Heuristic is not admissible
- 3c6** (-2 points) Incorrect derivation; heuristic happens to be admissible but not for the reasons given.

- 3c7** (-2 points) No explanation of why heuristic is admissible/reasonable.
3c8 (-5 points) Heuristic is not reasonable
3c9 (-1 points) Doesn't explain how to compute heuristic

4. [16 points] **Approximately Admissible A***

Assume we are given a heuristic function which is only approximately admissible. I.e., there exists some $\epsilon > 0$ such that for every node n in the tree, we have $h(n) \leq h^*(n) + \epsilon$. (Note that in this case it is possible for $h(n)$ for a *goal* node n to be bigger than zero.) Let n^* be the optimal goal node, with cost $g(n^*)$.

Assume that we run standard A* with our ϵ -admissible heuristic h . Let n_g be the goal node that A* returns. Prove that the cost $g(n_g)$ is at most $g(n^*) + \epsilon$. You may assume that the function $f = g + h$ corresponding to the heuristic function h is monotonic (and therefore that A* expands nodes ordered in increasing values of f). Hint: You may use any result which we have proved during the course. This should be about a five-line proof.

Answer: We have defined n^* to be the optimal goal node, such that $g(n^*) = c^*$. We want to show that the g -cost of the (potentially suboptimal) node n_g returned by A* is at most $c^* + \epsilon$.

As we showed in class, if f is monotonic then A* expands nodes in increasing f -value. Since A* returned n_g before n^* , we have

$$f(n_g) \leq f(n^*). \quad (1)$$

For every node n , the heuristic function $h(n) \leq h^*(n) + \epsilon$. Therefore:

$$\begin{aligned} f(n^*) &= g(n^*) + h(n^*) = c^* + h(n^*) \\ &\leq c^* + h^*(n^*) + \epsilon \\ &= c^* + \epsilon \quad \text{because } h^* \text{ of a goal is } 0 \end{aligned} \quad (2)$$

On the other hand, since $h(n_g) \geq 0$, we also know that

$$g(n_g) \leq g(n_g) + h(n_g) = f(n_g) \quad (3)$$

Combining the above equations, we get that

$$g(n_g) \leq f(n_g) \leq f(n^*) \leq c^* + \epsilon. \quad \blacksquare$$

Note: There were several common errors:

- Some people didn't explain why $f(n_g) \leq f(n^*)$. You needed to either mention monotonicity, talk about fringe nodes, or order of expansion in A*.
- When explaining $f(n_g) \leq f(n^*)$ using fringe nodes, many people correctly noted (as in the proof of A* optimality given in lecture) that we should compare to a node n along the shortest-cost path to n^* , hence wrote correctly that $f(n_g) \leq f(n) \leq f^*(n) + \epsilon$ (where $f^*(n) = g(n) + h^*(n)$). However, many incorrectly stated that because $f^*(n) = f^*(n^*)$ (often without explanation), $f(n_g) \leq f^*(n^*) + \epsilon = c^* + \epsilon$ holds. The above equality is not true; we can only state (by monotonicity of f) that $f^*(n) \leq f^*(n^*)$. Hence you were required to show that $f^*(n) \leq f^*(n^*) = c^*$; because this often meant that $h^*(n^*) = 0$ was not stated either, many will see this error marked as "E1 A3".

- Some people stated that $g(n_g) = f(n_g)$ instead of \leq . This assumes that $h(n_g) = 0$, which is not the case because h is not admissible (in fact it was pointed out in the first paragraph of the problem too). An “A4” error was marked for this.

Grading criteria:

A1 5 points for stating $f(n_g) \leq f(n^*)$ with a valid explanation.

A2 2 points for applying the ϵ -admissible heuristic $h(n) \leq h^*(n) + \epsilon$.

A3 3 points for identifying that $h^*(n^*) = 0$.

A4 2 points for stating that $g(n_g) \leq f(n_g)$ (deducted if equality stated, or stating that $h(n_g) = 0$).

- The remaining 4 points were ‘deduction’ points based on clarity of proofs:

E1 -2 points for some unclear explanations.

E2 -4 points for no explanation.

5. [12 points] **Constraint Satisfaction with Non-Binary Constraints**

In CSPs, a non-binary constraint is a constraint that involves more than 2 variables. For example, consider the variables X, Y, Z where the domain of X and Y is $\{Red, Green\}$ and the domain of Z is $\{Red, Green, Blue\}$. An example of a non-binary constraint is that exactly two of these variables have the color *Green*.

Sometimes we want to convert a non-binary CSP to a binary CSP. To do that we add a new extra variable for every constraint.

In detail, to replace a non-binary constraint C (over variables X_1, \dots, X_k) with only binary constraints, we create a new variable W . The variable W can take on as many values as there are legal assignments to X_1, \dots, X_k that satisfy C . We then remove the original constraint, and replace it with a set of binary constraints between the new variable W and the original variables X_i participating in the original constraint. For simplicity, you may assume that the original CSP does not contain any unary constraints.

- (a) [3 points] Complete the definition for a transformation of this type when it is executed for a constraint C over a set of variables X_1, \dots, X_k . You may assume that C is represented as a set of legal tuples of values for X_1, \dots, X_k . Specifically, how would you define W , and what are the binary constraints associated with W ?

Answer: Let C be the constraint, and S be the set of tuples of legal values for C . We introduce a new variable Y_C , which has a value y_s for each tuple s in S . Basically, the value of the variable Y_C dictates the values for all of the X_i 's. More precisely, for each $i = 1 \dots, k$, we introduce a binary constraint between Y_C and X_i :

$$\langle (Y_C, X_i), \{(y_s, x_i) \mid s \in S \text{ and } (i^{\text{th}} \text{ element of tuple } s) = x_i\} \rangle$$

Grading criteria:

- 1 points for correct domain of W .
 - 2 points for correct constraint definition.
 - -1 point for minor errors / vagueness
- (b) [3 points] Apply the algorithm to the example given above (where exactly two variables out of X, Y, Z have the color *Green*). Define exactly the domain of the new extra variable by writing down all the possible values in the domain. Show the new

constraints created by writing down all the constraints that involve Z and the new extra variable.

Answer: The new variable Y_C takes on values that are triples of colors, one for each of X, Y, Z . The set of legal triples for Y_C is:

$$\{(r, g, g), (g, r, g), (g, g, r), (g, g, b)\}$$

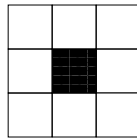
The set of legal values for the binary constraint involving Y_C and Z are:

Y_C	Z
(r, g, g)	g
(g, r, g)	g
(g, g, r)	r
(g, g, b)	b

Grading criteria:

- 1 points for correct domain of new variable.
- 2 points for constrain on Z .
- -1 for minor errors.

- (c) **[6 points]** Consider the problem of creating a crossword puzzle in which all the words must appear in some given dictionary D . We use the notation $D = \{w^1, w^2, \dots, w^n\}$, where w^i is the i -th word in the dictionary. Furthermore, we use the notation $w^i = w_1^i, w_2^i, \dots, w_{l_i}^i$ where w_j^i is the j -th letter of the i -th word and l_i is the length of w^i . As an example, consider the following instance of the problem:



Assuming that CAP,CAT,PEN,TIN $\in D$ (which can, of course, also contain other words), a possible solution could be:



- i. **[3 points]** Formalize this instance of the problem as a CSP problem, such that each variable corresponds to one empty square in the crossword puzzle (you will need 8 variables). What are the constraints in this formalization?

Answer: We have a variable X_i for each square in the crossword puzzle. In this particular case, we have 8 squares and 8 corresponding variables. We have a constraint for each word in the crossword puzzle. In our case, we would have four constraints. Consider some constraint C , and let X_{i_1}, \dots, X_{i_k} be the letters participating in the word, in the correct order. For example, let C be the constraint corresponding to the first horizontal word, and let's assume that we number the squares in our crossword from left to right and from top to bottom; in this case, the variables participating in C are X_1, X_2, X_3 . The set of legal tuples of values for C are those where $(x_{i_1}, \dots, x_{i_k}) \in D$. In our crossword puzzle, if CAP,

PEN, TIN, and CAT are the only legal words, then the set of legal values for C is $\{(C, A, P), (P, E, N), (T, I, N), (C, A, T)\}$.

Grading criteria:

- 1 point for defining variables correctly.
- 2 points for correctly defining constraint.
- -1 point for minor mistake or vagueness.

- ii. [**3 points**] Note that the constraints are non-binary. If we reformulate the problem as a binary CSP using the method from the previous parts, the extra variables and their domains have very intuitive meanings. What are they? (This answer should not take more than one sentence!)

Answer: In this case, the variables correspond to words in the crossword, their values to possible words in the dictionary (that have the right number of letters). This is an alternative, and very viable, formulation of a crossword puzzle as a CSP. In general, there is typically a duality in CSPs, in that we have a choice in what are the variables and what are the constraints.

Grading criteria:

- 0 points for incorrect binary-CSP transformation.
- 3 points for correct explanation.
- -1 point for vague / confusing answers.