Problem Solving

Techniques for visualizing non-trivial problems and coming up with solutions

Problem-Solving

- Computer Science is about problem solving.
  - Writing a solution for problems when the answer isn't immediately self evident.
- There isn't an exact prescribed formula or series of steps that you can learn and apply.
  - In that way it's similar to other tasks where you learn a basic skill set or are taught 'good' principles and you must determine how to apply them e.g. composing poems, writing short stories.

Problem-Solving (2)

- This section cannot/will not provide you with a simple set of steps that you can memorize and apply in any situation.
  - Think about the many ways in which your assignments could have been implemented!
- What will be provided is a set of approaches that may be used to solve some types of problems.
  - Afterwards you may be able to apply these techniques when faced with problems that you face in future.

  OR
  - You will be able to develop your own approaches.

James Tam

The Contents Of These Lectures May Look Unusual But…

- (An actual testimonial/thank you about the value of this section of the course – included here with permission).

  Hello Dr Tam,

  I just wanted to thank you for directing me to the problem solving lectures because I ended up pseudo coding the program and managed to fully implement method 2b within a few hours, I found those slides very useful and they helped me visualize the program a lot better.

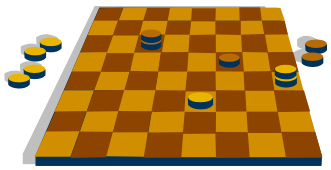
  Thank you for the help,

- …so don't "tune out" or 'bail' during these classes. (^_-)

James Tam

How To Come With An Algorithm/Solution

- An algorithm is the series of steps (not necessarily linear!) that provide the solution to your problem.
- "**Draw a picture**": If there is a physical analogy to the problem then try visualizing the problem using real world objects or scenarios.



How To Come With An Algorithm/Solution (2)

- If the problem is more abstract (e.g., mathematical and no obvious physical model can be created)
  - For simple problems this may not be an issue (you can immediately see the solution).
  - For more complex problems you may be unable to come with the general solution for the program.
  - **Extrapolate the specific to the general**: Try working out a solution for a particular example and see if that solution can be extended from that specific case to a more generalized formula.
    - If you can't get the general formula then try working out the solution to a few more specific examples.

Problem #1: Change Making

- (Paraphrased from the book "Pascal: An introduction to the Art and Science of Programming" by Walter J. Savitch.

Problem statement:

Design a program to make change. Given an amount of money, the program will indicate how many quarters, dimes and pennies are needed. The cashier is able to determine the change needed for values of a dollar or less.

Actions that may be needed:

•Action 1: Prompting for the amount of money

•Action 2: Computing the combination of coins needed to equal this amount

•Action 3: Output: Display the number of coins needed

Problem #1: Change Making

- How do you come up with the algorithm/solution for determining the amount of change owed?
  - **Don't just shout the solution!**
    - The solution is just the answer to one specific problem: which really isn't that important).
  - Instead think about *how to come up with* the solution
    - This is the ability to come up with a solution to ANY problem as opposed to just knowing the solution to ONE problem.

Problem #1: Change Making

- Problem solving strategies:
  - "Drawing a picture": no real physical analogy
  - **"Extrapolate the specific to the general":** In this case you can try working out the algorithm for a specific example (i.e., how much change do you give back when a specific amount is owed and extrapolate that specific case to the general formula which will work for any amount owed).
- Example amounts of change:
  - Work out change making for different but specific amounts e.g., $0.25, $0.26, $0.30, $0.32 etc.
  - Use these specific examples to help you discover the general formula for the problem (in this case making change).
- **Name of the folder containing the two pseudo code solutions**: `change_making`

$0.25

$0.01

$0.10

James Tam

Problem #2: Path Finding ("Chase Algorithm")

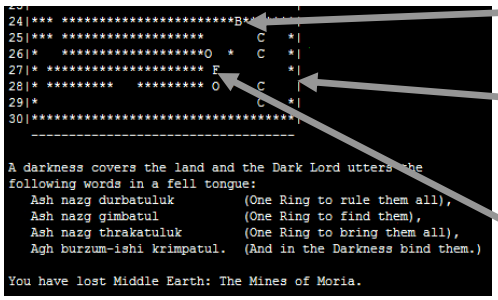
- Path-finding example: the computer program must find a path from "Point A" to "Point B" on the map.

Icewind Dale © Black Isle

James Tam

Problem #2: Path Finding ("Chase Algorithm")

- This problem is "too hard"? Path-finding was actually part of an old CPSC 231 assignment:
  - Make the Balrog: 'B' chase the Fellowship: 'F' (Balrog 'chases' or "finds the path towards" the Fellowship.



The 'Balrog': computer controlled

The exit

The 'Fellowship': human controlled

"Lord of the Rings" © George Allen & Unwin (Publishers)

James Tam

Problem #2: Handout

- Full game
  - The game creates a 4x5 world
  - Empty locations are set to a space ' ' while occupied locations consist either of 'P' (player) or 'M' (monster)
  - The 'player' is always located at (2,2)
  - The user is prompted for the starting location of the monster (no error checking under current version).
  - Based on the starting location for the player and the monster, the program will then move the monster towards (chase) the player.

James Tam

Problem #2: Students Do

- **Part I**: Assume that the destination (row, column) for the monster has already been determined, move the monster from the current location to the destination.
  - One example: assume these coordinates have been already determined by completing "Part II"
    - Monster source location (0,0), monster destination (1,1)
    - Make the monster 'disappear' off (0,0) and reappear on (1,1)
- **Part II**: Given that the player has specified the starting location for the monster write the code to determine the destination for the monster (move's towards the player).
  - The monster can only move to an adjacent square (one 'spot') when chasing the player.
  - The monster can move onto the same square as the player (appear to 'engulf' the player).

James Tam

Problem #2: Hint

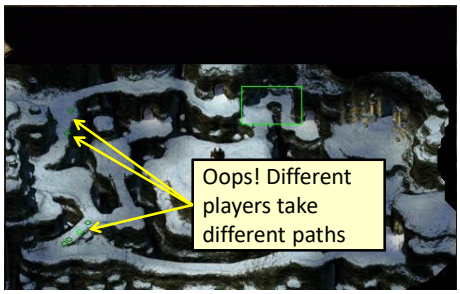
- What exactly does chase mean?
  - Move towards, get closer
- In order to determine what exactly 'move towards' means try:
  - Applying first problem solving technique "Draw a picture": Create a physical replica of the game world (numbered grid using a white/black board or graph paper).
  - Applying second problem solving technique "Extrapolate the specific to the general": Use specific examples of "chase scenarios" to help you determine the solution for any time a chase may occur.




James Tam

Path Finding / Chase Algorithm: Real Life

- One last note: the solution needed for this problem was fairly rudimentary.
  - Ignored the possibility of obstacles, didn't consider 'optimal' paths.
  - In real life path-finding can be very hard e.g., 'MapQuest', Apple 'Maps'



"Icewind Dale" © Black Isle

Optional extra video, a link to a commercial program that applies the chase algorithm:
https://www.youtube.com/watch?v=lbDJaCrcJ60&feature=youtu.be

James Tam

Problem Solving: Final Hint

- Keep in mind: a computer is dumb!
- Each step in an algorithm must be explicitly given to a computer.
  - Either the programmer writes the step in the form of a program instruction.
  - Or else the programming language includes this information as part of it's syntax.
  - When specifying a solution to a problem don't skip steps!

James Tam

Detail Level Required: Example

- What is *an algorithm* for finding the word in the following collection that comes first (alphabetical order):
  - Dog
  - Cat
  - Bird
  - Fish

James Tam

Specifying Details: Example

- …now find the word that comes first

- sometimes be more a nuisance than a benefit. This was found to be the case in my own investigation of potential change display mechanisms summarized in Chapter 5 and published as Tam, McCaffrey, Maurer, and Greenberg (2000). During this study, many test participants expressed a desire for useful abstractions that combine rudimentary change information into one higher-level conceptual change. For example, one participant noted while watching the animated replay of a class name being shown, "…I don't need to see each and every character being typed just to see a name change!" Of course, care must be taken to make these abstractions understandable, e.g., by using already familiar representations or notations. This minimizes the cost of acquiring information while maximizing its benefits due to the added structure and organization.
- Based upon my previous findings (to be discussed in Chapter 5), I add a third dimension, *persistence*, to Gutwin's classification. Persistence refers to how long the information is displayed (Figure 4.1 side pane). The display of information is *permanent* if it is always visible and *passing* if it only appears for a certain period. We noticed how study participants frequently complained when important information disappeared off the screen. Conversely, they also indicated that screen clutter might occur with the mechanisms that constantly displayed all changes. Thus, there's a need to classify change information according to how long it should stay visible.
- With permanent persistence, the effort needed to find changes i.e., the acquisition cost is low because the information is always there. Ideally, a person merely has to shift their gaze over to see the information. Because people can become accustomed to the occurrence of workspace events, they can also ignore things that do not interest them and pay closer attention to things that are of interest (Gutwin 1997).
- With passing persistence, information about changes is presented only for a limited duration. This is useful when the information applies only to a specific portion of the project (artifact or group of artifacts) being viewed, or when the change information otherwise becomes irrelevant. This is quite an important point for us.
- The matrix in Figure 4.1 suggests that these dimensions can be combined, giving eight possibilities. For example, a literal, situated and passing display of changes is depicted in Figure 4.2a. The figure shows an animation of a changed circle (by using a 'replay' technique) where the circle literally retraces the path that it took as it was moved. It is situated because the animation occurs in the same place that the change actually happened. The persistence is 'passing' because once an animation has replayed a change, the information is gone. Figure 4.2b shows two other examples within a concept map editor. The first illustrates the symbolic, situated and permanent octant, where color value (shades of gray) is used to indicate changed 'Jim' and 'Jack' nodes. Thus, it is symbolic because changes are mapped to a gray scale value, situated because the shading is applied directly to the node that was changed, and permanent because the color values are always on. Figure 4.2b also portrays an example of the symbolic, separate, and passing octant, where a person can raise a node's change details in a pop-up as a text description by mousing-over the node. Thus it is somewhat separate as the information appears outside the changed node, it is symbolic as it uses the text to describe the changes, and passing because the pop-up disappears when the person moves the mouse off the node (not quite on the node).

Exert from "**Supporting Change Awareness in Visual Workspaces"** (Tam 2002)

James Tam